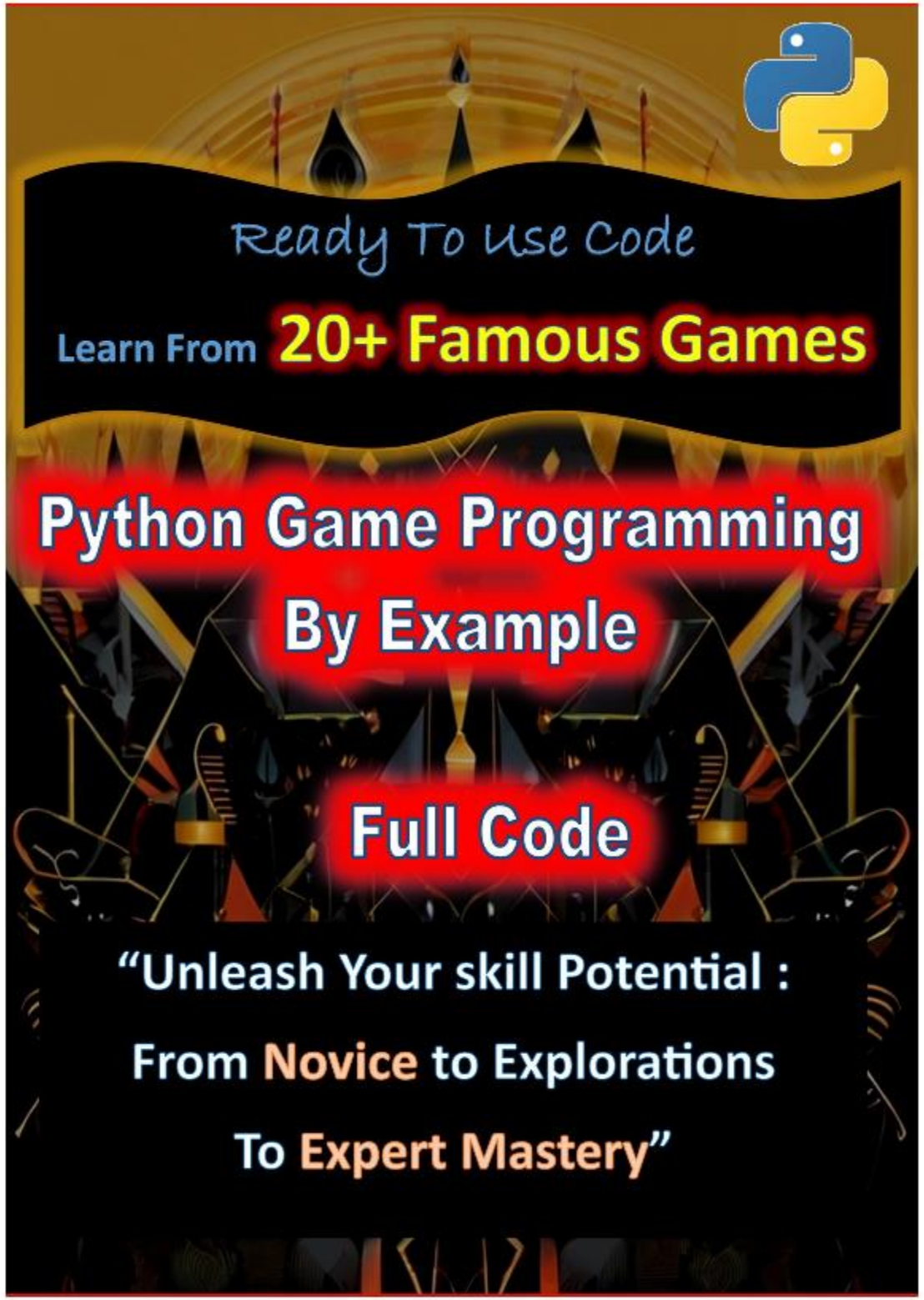

Ready To Use Code
Learn From 20+ Famous Games
Python Game Programming
By Example
Full Code
“Unleash Your skill Potential :
From Novice to Explorations
To Expert Mastery”

Table of Contents

1. GuessMaster: Number Adventure
    - Importing Libraries :
    - NumberGuessingGame_Class :
    - Main Script :
    - Summary :
- how to play GuessMaster: Number Adventure
2. Crossword Generator
- how to play_Crossword Generator
3. Hangman Game
    - HangmanGUI Class :
    - Main Function :
    - Usage :
    - Additional Notes :
- how to play hangman game
4. Tic Tac Toe Game
- How to Play Tic Tac Toe
    - Game Setup :
    - Gameplay :
    - Example Gameplay :
    - Winning Combinations :
    - Tips :

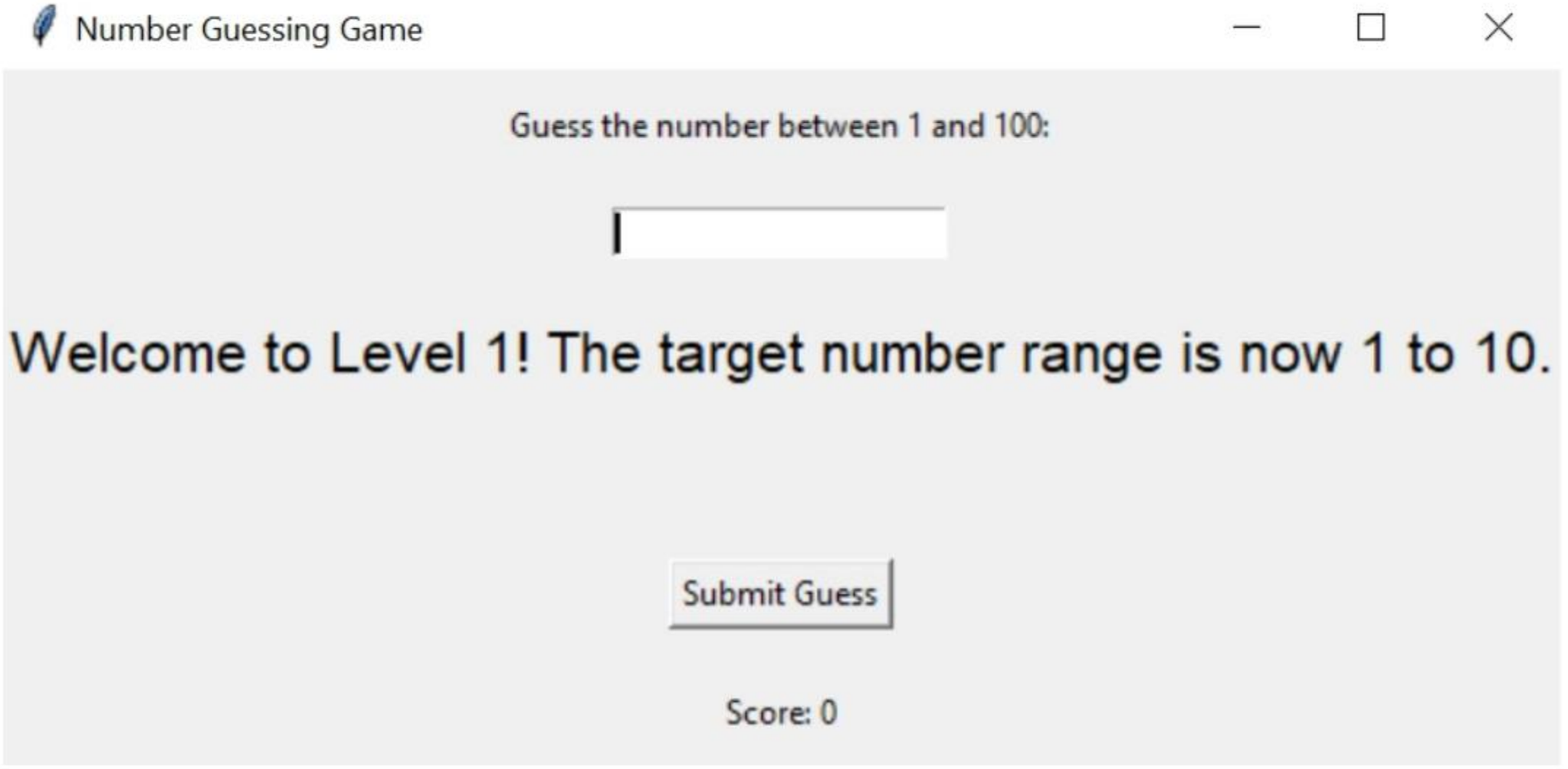
How To Play Maze Solver Game
6. Snake Game
How To Play Snake Game
7. Memory Puzzle Game
How To Play Memory Puzzle Game
8. Quiz Game
How To Play Quiz Game
9. 2048 Game
- Pygame Initialization :
- Constants and Configuration :
- Pygame Screen Initialization :
- Grid and Tile Drawing :
- Tile Colors and Themes :
- Tile Movement and Animation :
- Game State Management :
- Main Game Loop :
- Additional Features :
- Running the Game :

How To Play 2048 Game
10. BlackJack Game
- Import Statements :
- BlackjackGame Class :

Methods :
GUI Elements :
Main Function :
Overall Flow :
How To Play Blackjack Game
11. Soduku Solver Game
How To Play Sudoku Solver Game
12. Connect Four Game
How To Play Connect Four Game
13. Flappy Bird Clone Game
How To Play Flappy Bird Clone
14. Pong Game
How To Play Pong Game
15. Word Search Generator Game
How To Play Word Search Generator
16. Battleship Game
How To Play Battleship Game
17. Space Invader Game
How To Play Space Invader Game
18. Chess Game
How To Play Chess Game
19. Roulette Simulator Game
How To Play Roulette Simulator Game

20. Mancala Game
How To Play Mancala Game
21. Tower Defense game
How To Play Tower Defense Game
22. Sokuban Game
How To Play Sokoban Game
23. Breakout Game
How To Play Breakout Game
24. Sim City Clone Game
How To Play Sim City Clone Game
25. Simon Says Game
How To Play Simon Says Game
26. Ludo Game
How To Play Ludo Game

1. GuessMaster: Number Adventure




```
import tkinter as tk
from tkinter import messagebox
import random


class NumberGuessingGame:
```

```
def __init__(self, master):
    self.master = master
    self.master.title("Number Guessing Game")

    self.total_levels = 5
    self.level = 1
    self.target_number = self.generate_target_number()
    self.guesses_left = 10
    self.score = 0

    self.label = tk.Label(
        master, text="Guess the number between 1 and 100:")
    self.label.pack(pady=10)

    self.entry = tk.Entry(master)
    self.entry.pack(pady=10)

    self.level_label = tk.Label(
        master, text=f"Level: {self.level}", font=("Helvetica", 16))
    self.level_label.pack(pady=10)

    self.result_label = tk.Label(master, text="")
    self.result_label.pack(pady=10)
```

```
    self.submit_button = tk.Button(
        master, text="Submit Guess", command=self.check_guess)
    self.submit_button.pack(pady=10)

    self.reset_button = tk.Button(
        master, text="Play Again", command=self.reset_game)
    self.reset_button.pack(pady=10)
    self.reset_button.pack_forget()

    self.score_label = tk.Label(master, text=f"Score: {self.score}")
    self.score_label.pack(pady=10)

    self.start_level_message()

def generate_target_number(self):
    return random.randint(1, 10 * self.level)

def start_level_message(self):
    level_message = f"Welcome to Level {self.level}! The target number range is now 1 to {10 * self.level}."
    self.level_label.config(text=level_message)
    # Display for 3 seconds
    self.master.after(3000, self.clear_level_message)

def clear_level_message(self):
```

```
    self.entry.delete(0, tk.END)
    self.entry.focus_set()

def check_guess(self):
    user_input = self.entry.get()

    if not user_input:
        self.result_label.config(text="Please enter a valid guess.")
        return

    try:
        user_guess = int(user_input)
    except ValueError:
        self.result_label.config(text="Please enter a valid integer.")
        return

    if user_guess == self.target_number:
        self.result_label.config(
            text=f"Congratulations! You guessed the correct number. Score: {self.calculate_score()}")
        self.submit_button.config(state=tk.DISABLED)
        self.reset_button.pack(pady=10)

        if self.level < self.total_levels:
            # Display for 1 second before moving to the next level
```

```
            self.master.after(1000, self.next_level_message)
        else:
            self.show_performance_feedback()
    else:
        self.guesses_left -= 1
        if self.guesses_left == 0:
            self.result_label.config(
                text=f"Sorry, you're out of guesses. The correct number was {self.target_number}.")
            self.submit_button.config(state=tk.DISABLED)
            self.reset_button.pack(pady=10)
            self.show_performance_feedback()
        else:
            hint = "Too low. Try again!" if user_guess < self.target_number else "Too high. Try again!"
            self.result_label.config(
                text=f"Incorrect! {hint} Guesses left: {self.guesses_left}")
            self.update_score_label()

def calculate_score(self):
    score = self.guesses_left * 10 * self.level
    self.score += score
    self.update_score_label()
    return self.score

def update_score_label(self):
```

```
    self.score_label.config(text=f"Score: {self.score}")

def next_level_message(self):
    self.level += 1
    self.target_number = self.generate_target_number()
    self.guesses_left = 10
    self.submit_button.config(state=tk.NORMAL)
    self.level_label.config(text=f"Level: {self.level}")
    self.start_level_message()

def show_performance_feedback(self):
    feedback = f"All levels completed! Your total score is {self.score}."
    messagebox.showinfo("Game Over", feedback)

def reset_game(self):
    self.level = 1
    self.target_number = self.generate_target_number()
    self.guesses_left = 10
    self.label.config(text="Guess the number between 1 and 100:")
    self.result_label.config(text="")
    self.submit_button.config(state=tk.NORMAL)
    self.reset_button.pack_forget()
    self.score = 0
    self.update_score_label()
```

```
    self.level_label.config(text=f"Level: {self.level}")
    self.start_level_message()
```

```
if __name__ == "__main__":
    root = tk.Tk()
    game = NumberGuessingGame(root)
    root.mainloop()
```

This Python script creates a simple number guessing game using the Tkinter library for the graphical user interface (GUI). Let's break down the code step by step :

Importing Libraries :

- tkinter : This library is used for creating GUI applications .
- random : This library is used for generating random numbers .

NumberGuessingGame Class :

This class represents the main functionality of the game .

Constructor (__init__):

- Initializes the game parameters such as total_levels , level , target_number , guesses_left , and score .
- Sets up the GUI elements including labels, entry fields, buttons, etc .

- Calls start_level_message () method to display a welcome message for the first level .

generate_target_number () Method :

- Generates a random target number within a specific range based on the current level .

start_level_message () Method :

- Displays a welcome message for the current level .
- Clears the message after 3 seconds using after () method .

clear_level_message () Method :

- Clears the entry field after displaying the level message .

check_guess () Method :

- Checks the user's guess against the target number .
- Handles cases where the input is invalid, the guess is correct, or the guess is incorrect .
- Updates the GUI accordingly .

calculate_score () Method :

- Calculates the score based on the remaining guesses and the current level .
- Updates the total score and the score label .

update_score_label () Method :

- Updates the score label with the current score .

next_level_message () Method :

- Prepares for the next level by updating parameters and generating a new target number .
- Displays a message for the next level .

show_performance_feedback () Method :

- Shows a message box with the performance feedback when all levels are completed .

reset_game () Method :

- Resets the game parameters to start a new game .
- Clears the GUI elements and sets up for the first level .

Main Script :

- Creates a Tkinter `Tk` instance .
- Creates an instance of `NumberGuessingGame` class passing the `Tk` instance .
- Starts the Tkinter event loop using `mainloop ()` method .

Summary :

This script creates a GUI - based number guessing game where the player has to guess a randomly generated number within a specific range . The game consists of multiple levels, and the player's score is based on the number of guesses left and the current level . The game interface provides feedback to the player on each guess and displays the final performance feedback upon completing all levels . Additionally, the player can play again after completing the game or resetting the game at any point .

how to play GuessMaster: Number Adventure

1. **Launching the Game :**
    - Run the Python script provided .
    - A window will appear with the title " Number Guessing Game " and the initial level message .
2. **Reading the Level Message :**
    - The level message informs you about the target number range for the current level (e . g . , " Welcome to Level 1 ! The target number range is now 1 to 10 .").
3. **Making a Guess :**

- Enter your guess into the entry field .
- Click the " Submit Guess " button .

4. **Receiving Feedback :**

- If your guess is correct, you'll receive a congratulatory message and your score for that round .
- If your guess is incorrect, you'll receive a hint （too low or too high）, the remaining guesses, and your score will be updated .

5. **Advancing to the Next Level :**

- If you guess correctly within the allowed guesses, you'll automatically advance to the next level after a short delay .
- The game will update the target number range for the new level .

6. **Game Over :**

- The game ends when you complete all levels or run out of guesses .
- If you complete all levels, a message box will appear with your total score .

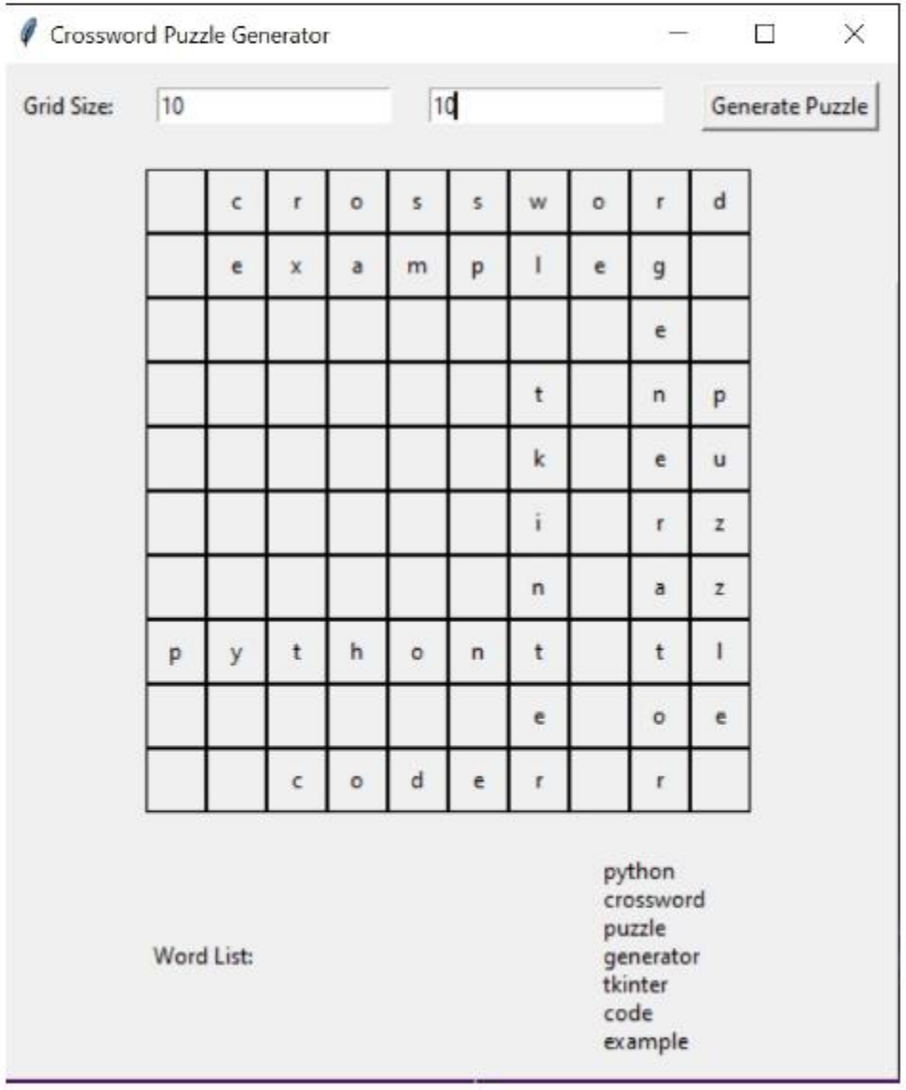
7. **Playing Again :**

- Click the " Play Again " button to reset the game and start from Level 1 .

Tips :

- Pay attention to the level message for the updated target number range .
- Try to guess the correct number within the given number of attempts to maximize your score .
- If you run out of guesses, the correct number will be revealed, and you can choose to play again .

2. Crossword Generator



```
import tkinter as tk
from tkinter import messagebox
import random


class CrosswordGenerator:
  def __init__(self, root):
    self.root = root
    self.root.title("Crossword Puzzle Generator")


    self.grid_size_label = tk.Label(root, text="Grid Size:")
    self.grid_size_label.grid(row=0, column=0, padx=10, pady=10)

    self.rows_entry = tk.Entry(root)
    self.rows_entry.grid(row=0, column=1, padx=10, pady=10)

    self.cols_entry = tk.Entry(root)
    self.cols_entry.grid(row=0, column=2, padx=10, pady=10)

    self.generate_button = tk.Button(
      root, text="Generate Puzzle", command=self.generate_puzzle)
```

```
21.     self.generate_button.grid(row=0, column=3, padx=10, pady=10)
22.
23.     self.puzzle_frame = tk.Frame(root)
24.     self.puzzle_frame.grid(row=1, column=0, columnspan=4, padx=10, pady=10)
25.
26.     self.word_list_label = tk.Label(root, text="Word List:")
27.     self.word_list_label.grid(
28.       row=2, column=0, padx=10, pady=10, columnspan=2)
29.
30.     self.word_list_var = tk.StringVar()
31.     self.word_list_display = tk.Label(
32.       root, textvariable=self.word_list_var, wraplength=200, justify="left")
33.     self.word_list_display.grid(
34.       row=2, column=2, padx=10, pady=10, columnspan=2)
35.
36.   def generate_puzzle(self):
37.     try:
38.       rows = int(self.rows_entry.get())
39.       cols = int(self.cols_entry.get())
40.
```

```
41.         if rows <= 0 or cols <= 0:
42.             messagebox.showerror(
43.                 "Error", "Grid size should be positive integers.")
44.             return
45.
46.         puzzle, word_list = self.create_puzzle(rows, cols)
47.         self.display_puzzle(puzzle)
48.         self.display_word_list(word_list)
49.
50.     except ValueError:
51.         messagebox.showerror(
52.             "Error", "Please enter valid integers for grid size.")
53.
54. def create_puzzle(self, rows, cols):
55.     puzzle = [[' ' for _ in range(cols)] for _ in range(rows)]
56.
57.     words = ["python", "crossword", "puzzle", "generator", "tkinter", "code", "example"]
58.
59.     word_list = "\n".join(words)
60.
```

```
    for word in words:
        direction = random.choice(['across', 'down'])
        placed = False


        for _ in range(10):  # Try placing the word multiple times
            if direction == 'across':
                row = random.randint(0, rows - 1)
                col = random.randint(0, cols - len(word))
                if all(puzzle[row][col + i] == ' ' for i in range(len(word))):
                    for i in range(len(word)):
                        puzzle[row][col + i] = word[i]
                    placed = True
                    break
            else:
                row = random.randint(0, rows - len(word))
                col = random.randint(0, cols - 1)
                if all(puzzle[row + i][col] == ' ' for i in range(len(word))):
                    for i in range(len(word)):
                        puzzle[row + i][col] = word[i]
                    placed = True
                    break


        if not placed:
```

```
            messagebox.showwarning("Warning", f"Unable to place the word '{word}' in the puzzle.")


    return puzzle, word_list




  def display_puzzle(self, puzzle):
    for widget in self.puzzle_frame.winfo_children():
      widget.destroy()


    for i, row in enumerate(puzzle):
      for j, cell in enumerate(row):
        label = tk.Label(self.puzzle_frame, text=cell,
                         width=4, height=2, relief="solid", borderwidth=1)
                                                label.grid(row=i, column=j)


```

```
102.                 def display_word_list(self, word_list):
103.                     self.word_list_var.set(word_list)


104.
105.             if __name__ == "__main__":
106.                 root = tk.Tk()
107.                 crossword_generator = CrosswordGenerator(root)
108.                 root.mainloop()
```

This Python script creates a simple Crossword Puzzle Generator using the Tkinter library . The program has a graphical user interface （GUI） with input fields to specify the grid size and a button to generate a crossword puzzle . Let's break down the components of the script :

1. **Imports :**
    - tkinter : The standard GUI toolkit for Python .
    - messagebox : A submodule of Tkinter used for displaying various types of message boxes .
    - random : Used for generating random numbers and choices .
2. **Class : CrosswordGenerator**
    - **Initialization (__init__):**

- Initializes the Tkinter window (root) and sets its title .
- Creates various GUI elements such as labels, entry fields, buttons, and a frame to display the crossword puzzle .

- **Method : generate_puzzle :**
  - Retrieves the grid size (number of rows and columns) entered by the user .
  - Validates the input to ensure positive integers are provided .
  - Calls the create_puzzle method to generate a crossword puzzle and displays it along with the word list .
- **Method : create_puzzle :**
  - Takes the number of rows and columns as input and initializes an empty grid .
  - **Def**ines a list of words to be used in the crossword .
  - Randomly selects a direction ('across' or 'down') for each word and attempts to place it on the grid . The placement is tried multiple times (up to 10) if unsuccessful .
  - If a word cannot be placed, a warning message is displayed .
  - Returns the generated puzzle grid and the formatted word list .
- **Method : display_puzzle :**

- Clears the existing widgets in the puzzle frame .
- Iterates through the puzzle grid and creates Tkinter labels for each cell, displaying the characters in a grid layout .

- **Method : display_word_list :**
  - Sets the Tkinter StringVar (word_list_var) with the formatted word list .

3. **Main Block :**
   - Creates a Tkinter root window and an instance of the CrosswordGenerator class .
   - Enters the Tkinter event loop (root . mainloop ()) to handle user interactions .
4. **Example Words :**
   - The script uses a pre**def**ined list of words for the crossword puzzle . You can modify the words list in the create_puzzle method to customize the words used .
5. **Widgets and Layout :**
   - The GUI includes labels, entry fields, and buttons organized in a grid layout to allow user input and puzzle display .
6. **Note :**

- The script has some basic error handling to ensure valid input and warns the user if a word cannot be placed in the puzzle .

This script provides a simple demonstration of a crossword puzzle generator with a graphical interface using Tkinter . You can further enhance and customize the functionality based on your requirements .

how to play Crossword Generator

The generated crossword puzzle is not interactive in the current implementation, meaning you can't directly interact with it by clicking on cells or typing letters . However, I can provide you with a simple guide on how you might play a generated crossword puzzle :

1. **Generate Puzzle :**
    - Run the script and provide the desired grid size （ number of rows and columns ） when prompted .
    - Click the " Generate Puzzle " button .
2. **View Puzzle :**
    - The crossword puzzle grid and word list will be displayed in the Tkinter window .
3. **Understand the Display :**
    - The puzzle grid is displayed in the Tkinter window with each cell containing a single letter . Empty cells are represented by spaces .
    - The word list is displayed on the right side of the window, showing the words that need to be found in the puzzle .
4. **Manually Solve :**

- You can manually solve the puzzle by looking at the words in the list and entering them into the corresponding cells in the grid .

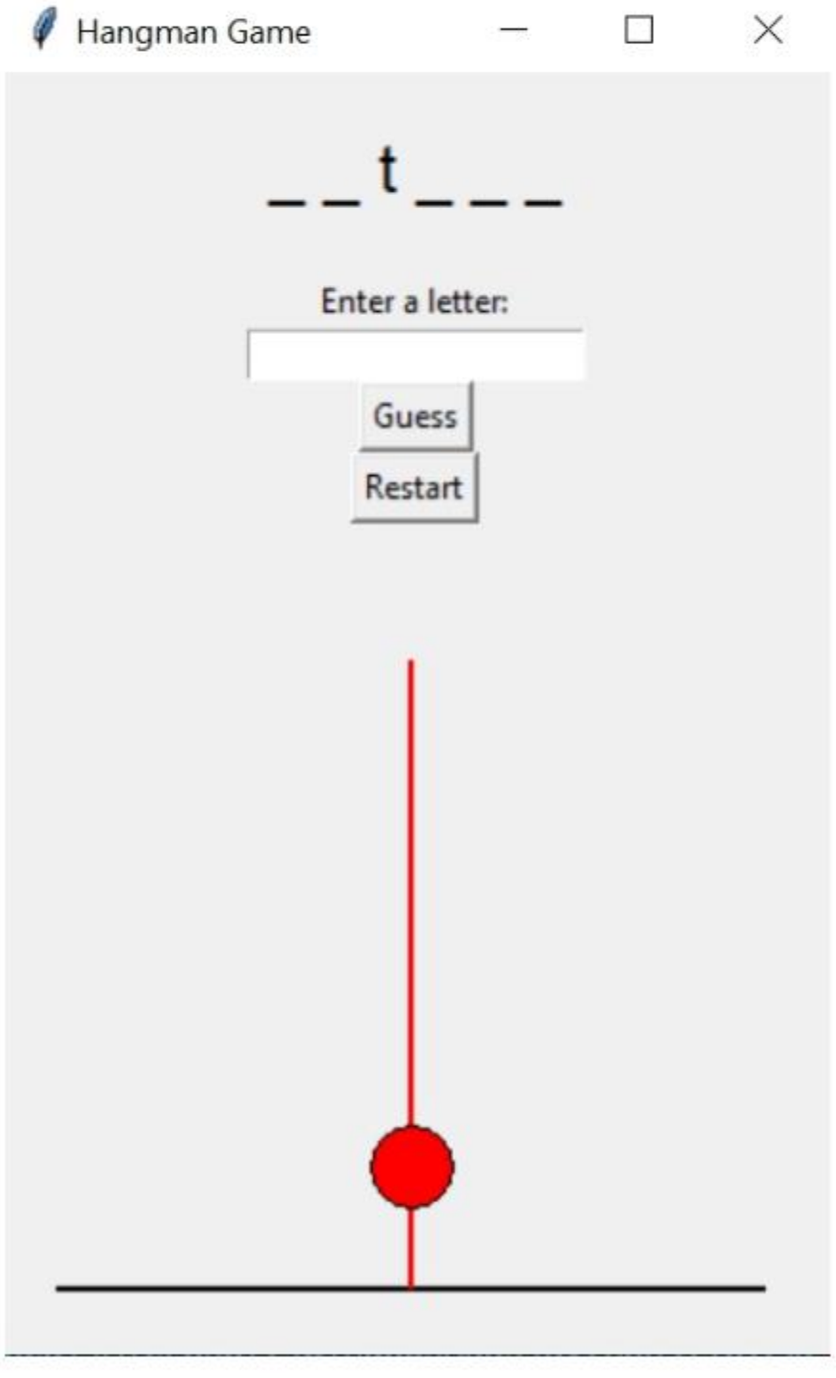
5. **Check Placement :**

- Crossword puzzles typically follow certain rules, such as words intersecting at a common letter . Ensure that your entries respect these rules .

6. **Verify Correctness :**

- Compare your filled - in puzzle with the word list to ensure you've correctly placed all the words .

3. Hangman Game



```
import tkinter as tk
from tkinter import messagebox
import random
eye_radius = 4
mouth_radius = 8
head_y=0
class HangmanGUI:
  def __init__(self, master):
    self.master = master
    self.master.title("Hangman Game")



    self.word_list = ["python", "hangman", "programming",
              "computer", "developer", "algorithm", "coding"]
    self.word_to_guess = ""
    self.guesses = set()
    self.max_attempts = 6
    self.attempts_left = self.max_attempts
    self.head_radius = 15



    self.word_label = tk.Label(self.master, text="", font=("Arial", 18))
    self.word_label.pack(pady=20)
```

```
        self.guess_label = tk.Label(self.master, text="Enter a letter:")
        self.guess_label.pack()

        self.guess_entry = tk.Entry(self.master)
        self.guess_entry.pack()

        self.guess_button = tk.Button(
            self.master, text="Guess", command=self.make_guess)
        self.guess_button.pack()

        self.restart_button = tk.Button(
            self.master, text="Restart", command=self.restart_game)
        self.restart_button.pack()

        self.draw_canvas()

        self.choose_word()
        self.update_word_label()

    def draw_canvas(self):
```

```
        self.canvas = tk.Canvas(self.master, width=300, height=300)
        self.canvas.pack()


        # Draw static pole and base
        self.canvas.create_line(150, 50, 150, 280, width=2)  # Pole
        self.canvas.create_line(20, 280, 280, 280, width=2)  # Base


    def choose_word(self):
        self.word_to_guess = random.choice(self.word_list)


    def update_word_label(self):
        display = ""
        for letter in self.word_to_guess:
            if letter in self.guesses:
                display += letter + " "
            else:
                display += "_ "
        self.word_label.config(text=display.strip())


    def make_guess(self):
        guess = self.guess_entry.get().lower()
```

```
        if guess.isalpha() and len(guess) == 1:
            if guess in self.guesses:
                messagebox.showinfo(
                    "Already Guessed", f"You have already guessed the letter '{guess}'.")
            else:
                self.guesses.add(guess)
                if guess not in self.word_to_guess:
                    self.attempts_left -= 1
                    self.draw_hangman()


                self.update_word_label()


                if self.attempts_left == 0:
                    self.game_over()
                elif "_" not in self.word_label.cget("text"):
                    self.game_win()
        else:
            messagebox.showinfo(
                "Invalid Input", "Please enter a valid single letter.")
```

```
        self.guess_entry.delete(0, tk.END)


    def draw_hangman(self):
        # Clear the canvas before drawing
        self.canvas.delete("all")


        # Draw static pole and base
        self.canvas.create_line(150, 50, 150, 280, width=2)  # Pole
        self.canvas.create_line(20, 280, 280, 280, width=2)  # Base


        if self.attempts_left < self.max_attempts:
            # Calculate the position of the head
            rope_bottom = 280
            max_head_y = rope_bottom - self.head_radius
            min_head_y = 50


            # Calculate head position based on remaining attempts
            head_y = max(min_head_y, max_head_y -
                         (self.max_attempts - self.attempts_left) * 30)
```

```
        # Draw the rope and head
        self.canvas.create_line(
          150, 50, 150, rope_bottom, width=2, fill="red") # Draw the rope
        self.canvas.create_oval(
                    150 - self.head_radius, head_y - self.head_radius, 150 + self.head_radius, head_y +
self.head_radius, fill="red") # Draw the head


        # Disable the guess button when the head reaches the top of the pole
        if head_y == min_head_y:
          self.guess_button.config(state=tk.DISABLED)


        # If the head is at the top, draw eyes and a sad mouth
        if head_y == min_head_y:




          # Draw eyes
          eye_x_left = 150 - 8
          eye_y = head_y - 6
          self.canvas.create_oval(
            eye_x_left - eye_radius, eye_y - eye_radius,
            eye_x_left + eye_radius, eye_y + eye_radius, fill="black") # Left eye
```

```
            eye_x_right = 150 + 8
            self.canvas.create_oval(
              eye_x_right - eye_radius, eye_y - eye_radius,
              eye_x_right + eye_radius, eye_y + eye_radius, fill="black")  # Right eye


            # Draw a sad mouth
            mouth_x = 150
            mouth_y = head_y + 10
            self.canvas.create_line(
              mouth_x - mouth_radius, mouth_y,
              mouth_x + mouth_radius, mouth_y, fill="black")


        elif not hasattr(self, 'game_over_shown'):
          # Draw the head on the top of the pole in red
          self.canvas.create_oval(
                150 - self.head_radius, min_head_y - self.head_radius, 150 + self.head_radius, min_head_y +
self.head_radius, fill="red")


          # Display a message about running out of attempts (once)
          self.game_over()
```

```
        self.guess_button.config(state=tk.DISABLED)
        self.game_over_shown = True


    def game_over(self):



        # Draw eyes
        eye_x_left = 150 - 8
        eye_y = head_y - 6
        self.canvas.create_oval(
            eye_x_left - eye_radius, eye_y - eye_radius,
            eye_x_left + eye_radius, eye_y + eye_radius, fill="black")  # Left eye

        eye_x_right = 150 + 8
        self.canvas.create_oval(
            eye_x_right - eye_radius, eye_y - eye_radius,
            eye_x_right + eye_radius, eye_y + eye_radius, fill="black")  # Right eye


        # Draw a sad mouth
        mouth_x = 150
```

```
        mouth_y = head_y + 10
        self.canvas.create_line(
            mouth_x - mouth_radius, mouth_y,
            mouth_x + mouth_radius, mouth_y, fill="black")



    def game_win(self):
        messagebox.showinfo("Congratulations",
                            "Congratulations! You guessed the word.")


    def restart_game(self):
        # Reset game state
        self.choose_word()
        self.guesses = set()
        self.attempts_left = self.max_attempts
        self.guess_button.config(state=tk.NORMAL)
        self.update_word_label()


        # Clear the canvas
        self.canvas.delete("all")

```

```
        # Draw static pole and base
        self.canvas.create_line(150, 50, 150, 280, width=2) # Pole
        self.canvas.create_line(20, 280, 280, 280, width=2) # Base



def main():
    root = tk.Tk()
    hangman_game = HangmanGUI(root)
    root.mainloop()



if __name__ == "__main__":
    main()
```

The provided **Python Code** is an implementation of a simple Hangman game using the Tkinter library for the graphical user interface . Let's go through the code in detail :

HangmanGUI Class :

1. **Initialization (__init__ method):**
    - The constructor initializes the main aspects of the game .
    - Sets up the Tkinter window and basic elements like labels, entry fields, buttons, and a canvas for drawing .

- **Def**ines attributes such as the word list, word to guess, guessed letters, maximum attempts, attempts left, and head radius .

2. **draw_canvas Method :**

- Initializes and draws the static parts of the hangman (pole and base) on the canvas .

3. **choose_word Method :**

- Randomly selects a word from the pre**def**ined word list .

4. **update_word_label Method :**

- Updates the word label based on the guessed letters, showing underscores for unguessed letters .

5. **make_guess Method :**

- Retrieves the guessed letter from the entry field .
- Checks if the guess is valid (a single alphabetical character) and whether it has been guessed before .
- Updates the game state, checks for a win or loss, and handles invalid inputs .

6. **draw_hangman Method :**

- Draws the hangman figure based on the number of attempts left .
- Handles the progression of the hangman figure as incorrect guesses are made .

7. **game_over Method :**

- Displays a game over message and draws a sad face when the player runs out of attempts .

8. **game_win Method :**

- Displays a congratulations message when the player successfully guesses the word .

9. **restart_game Method :**

- Resets the game state for a new round .
- Clears the canvas and draws the static hangman elements .

Main Function :

- Creates a Tkinter root window and initializes the HangmanGUI class .
- Starts the Tkinter main loop to run the GUI .

Usage :

- The player interacts with the game by entering a single letter in the entry field and clicking the " Guess " button .
- The canvas displays the hangman figure, and the word to guess is shown with underscores for unguessed letters .
- The game ends when the player either correctly guesses the word or runs out of attempts .

Additional Notes :

- The game uses a pre**def**ined word list, and a new word is chosen for each round .
- The maximum attempts are set to 6 by **def**ault, and the hangman figure progressively appears as incorrect guesses are made .
- The game provides options to restart and play again after a win or loss .

Make sure you have Tkinter installed (import tkinter as tk) before running this code . You can run it as a Python script to play the Hangman game in a simple GUI window .

how to play hangman game

To play the Hangman game using the provided code, follow these steps :

1. **Run the Code :**
    - Copy the provided **Python Code** into a Python environment or script .
    - Make sure you have Tkinter installed (it's usually included with Python installations).
2. **Execute the Script :**
    - Run the script .
    - A window will appear with the Hangman game GUI .
3. **Game Start :**

- The game will start by choosing a random word from the pre**def**ined word list .

4. **Guess a Letter :**

- Enter a single letter in the entry field provided .
- Click the " Guess " button .

5. **Game Progress :**

- The word to guess will be displayed with underscores for unguessed letters .
- The hangman figure will appear on the canvas based on incorrect guesses .

6. **Continue Guessing :**

- Keep entering letters and clicking the " Guess " button .
- The game will update the word display and hangman figure accordingly .

7. **Winning :**

- If you correctly guess the entire word, a congratulatory message will appear .

8. **Losing :**

- If you run out of attempts, a game over message will be displayed, and a sad face will appear .

9. **Restart :**

- You can restart the game by clicking the " Restart " button .
- A new word will be chosen, and the game state will be reset .

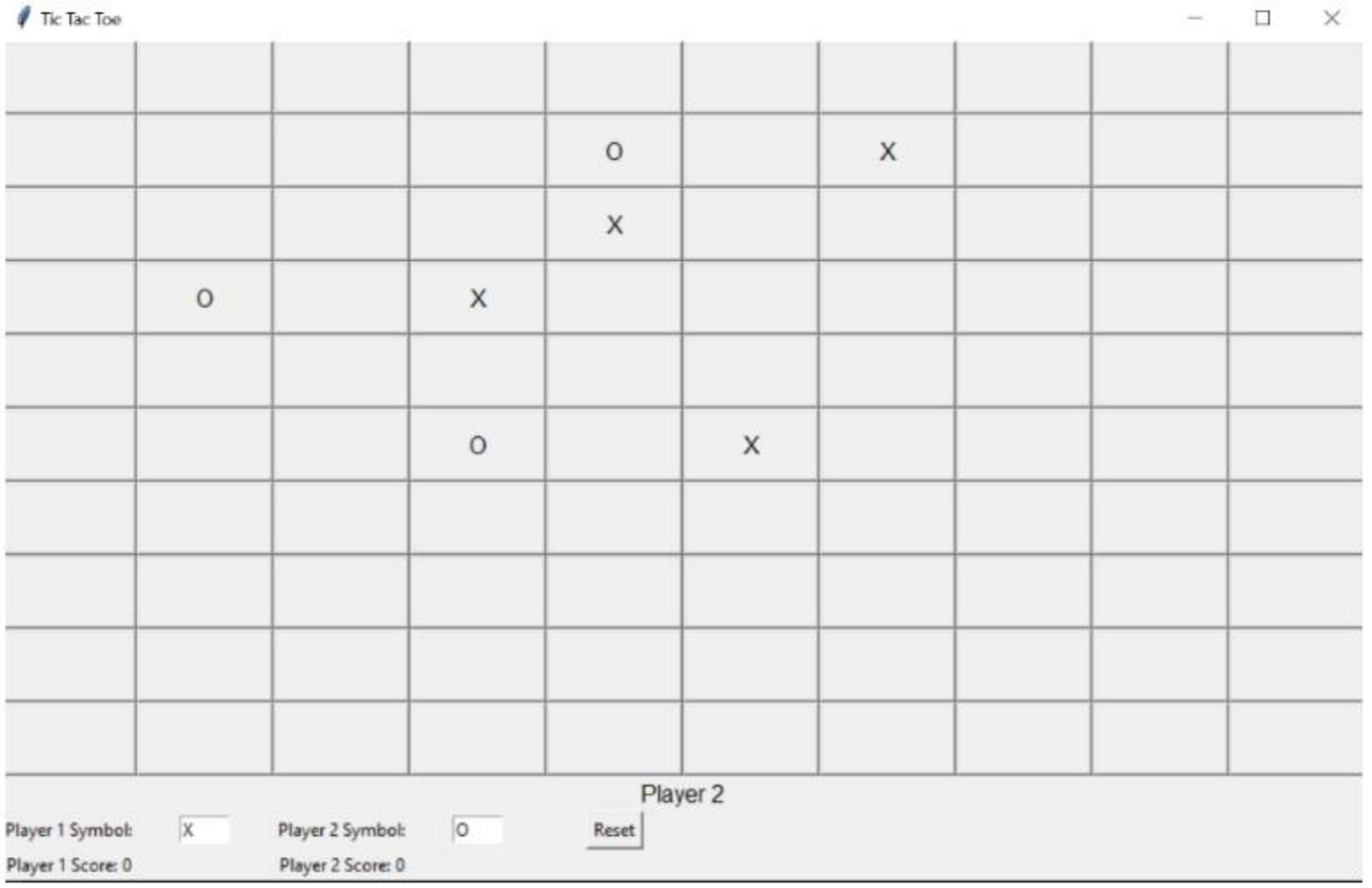
10. **Repeat :**

- Continue playing by guessing letters and trying to guess the word .

Remember :

- Only enter a single alphabetical letter as a guess .
- The game tracks guessed letters, and entering the same letter again will show a message .
- The maximum attempts are set to 6, and the hangman figure will progressively appear as you make incorrect guesses .

4. Tic Tac Toe Game




```
import tkinter as tk
from tkinter import messagebox

```

```
class TicTacToe:
  def __init__(self):
    self.window = tk.Tk()
```

```
    self.window.title("Tic Tac Toe")
    self.board_size = 10
    self.board = [[" for _ in range(self.board_size)]
                  for _ in range(self.board_size)]

    self.current_player = tk.StringVar()
    self.current_player.set("Player 1")
    self.player1_symbol = tk.StringVar()
    self.player2_symbol = tk.StringVar()
    self.player1_symbol.set("X")
    self.player2_symbol.set("O")
    self.player1_score = 0
    self.player2_score = 0

    self.create_widgets()

def create_widgets(self):
    for i in range(self.board_size):
        for j in range(self.board_size):
            btn = tk.Button(self.window, text='', font=('normal', 12), width=5, height=2,
                            command=lambda row=i, col=j: self.on_button_click(row, col))
            btn.grid(row=i, column=j, padx=0, pady=0, sticky="nsew")
            self.board[i][j] = btn
```

```
for i in range(self.board_size):
    self.window.grid_rowconfigure(i, weight=1, uniform="row")
    self.window.grid_columnconfigure(i, weight=1, uniform="col")

tk.Label(self.window, textvariable=self.current_player, font=(
    'normal', 12)).grid(row=self.board_size, columnspan=self.board_size)
tk.Label(self.window, text="Player 1 Symbol:").grid(
    row=self.board_size + 1, column=0)
tk.Entry(self.window, textvariable=self.player1_symbol,
         width=5).grid(row=self.board_size + 1, column=1)
tk.Label(self.window, text="Player 2 Symbol:").grid(
    row=self.board_size + 1, column=2)
tk.Entry(self.window, textvariable=self.player2_symbol,
         width=5).grid(row=self.board_size + 1, column=3)

self.reset_button = tk.Button(
    self.window, text="Reset", command=self.reset_board)
self.reset_button.grid(row=self.board_size + 1, column=4)
self.player1_score_label = tk.Label(
    self.window, text="Player 1 Score: 0")
self.player1_score_label.grid(row=self.board_size + 2, column=0)
self.player2_score_label = tk.Label(
    self.window, text="Player 2 Score: 0")
self.player2_score_label.grid(row=self.board_size + 2, column=2)
```

```
def on_button_click(self, row, col):
    if self.board[row][col]["text"] == '':
        self.board[row][col]["text"] = self.player1_symbol.get(
        ) if self.current_player.get() == "Player 1" else self.player2_symbol.get()
        if self.check_winner(row, col):
            messagebox.showinfo(
                "Game Over", f"{self.current_player.get()} wins!")
            self.update_scores()
            self.reset_board()
        elif self.check_draw():
            messagebox.showinfo("Game Over", "It's a draw!")
            self.reset_board()
        else:
            self.switch_player()

def check_winner(self, row, col):
    symbol = self.board[row][col]["text"]

    # Check row
    if all(self.board[row][c]["text"] == symbol for c in range(self.board_size)):
        return True
    # Check column
    if all(self.board[r][col]["text"] == symbol for r in range(self.board_size)):
```

```
            return True
        # Check diagonals
        if row == col and all(self.board[i][i]["text"] == symbol for i in range(self.board_size)):
            return True
        if row + col == self.board_size - 1 and all(self.board[i][self.board_size - 1 - i]["text"] == symbol for i in range(self.board_size)):
            return True
        return False

    def check_draw(self):
        for i in range(self.board_size):
            for j in range(self.board_size):
                if self.board[i][j]["text"] == '':
                    return False
        return True

    def switch_player(self):
        self.current_player.set(
            "Player 2" if self.current_player.get() == "Player 1" else "Player 1")

    def update_scores(self):
        if self.current_player.get() == "Player 1":
            self.player1_score += 1
            self.player1_score_label.config(
```

```
            text=f"Player 1 Score: {self.player1_score}")
        else:
            self.player2_score += 1
            self.player2_score_label.config(
                text=f"Player 2 Score: {self.player2_score}")

    def reset_board(self):
        for i in range(self.board_size):
            for j in range(self.board_size):
                self.board[i][j]["text"] = ''
        self.current_player.set("Player 1")
```

```
if __name__ == "__main__":
    game = TicTacToe()
    game.window.mainloop()
```

This is a simple implementation of the classic Tic Tac Toe game using the Tkinter library in Python . Let's break down the code and understand each component :

1. **Class Definition** (TicTacToe):
    - The TicTacToe class is initialized with the creation of a Tkinter window .
    - It sets the title of the window to " Tic Tac Toe " and **def**ines the size of the game board (10x10) and initializes the board with an empty state .

- Various attributes are **def**ined, such as current_player (to track the current player) , player1_symbol , player2_symbol , and scores for both players .
- The create_widgets method is responsible for setting up the GUI elements, including the game board buttons, labels, and entry fields for customizing player symbols .
- Buttons in the game board are created using nested loops, and their click events are connected to the on_button_click method .
- Labels are set up to display the current player, player symbols, and player scores .
- A reset button (reset_button) is created to reset the board .

2. **Event Handling** (on_button_click):

- This method is called when a button on the game board is clicked .
- Checks if the clicked button is empty; if so, it updates the button text with the current player's symbol .
- Checks for a winner using the check_winner method . If a winner is found, a message box is displayed, scores are updated, and the board is reset .
- If the game is a draw (no winner and no empty spaces left) , it shows a message box and resets the board .
- If the game continues, it switches to the next player .

3. **Winner Checking (check_winner):**

- This method checks for a winner by examining the current state of the board after each move .
- It checks rows, columns, and diagonals to see if all elements match the symbol of the current player .

4. **Draw Checking (check_draw):**

- This method checks for a draw by examining whether there are any empty spaces left on the board .

5. **Player Switching (switch_player):**

- This method switches the current player between " Player 1 " and " Player 2 ."

6. **Score Updating (update_scores):**

- This method updates the scores and the corresponding labels when a player wins .

7. **Board Resetting (reset_board):**

- This method resets the entire game board, setting all button texts to empty and resetting the current player to " Player 1 ."

8. **Main Block :**

- An instance of the `TicTacToe` class is created, and the Tkinter main loop is started with `game.window.mainloop()`.

Overall, this code provides a graphical user interface for playing Tic Tac Toe with customizable player symbols and keeps track of player scores.

How to Play Tic Tac Toe

Tic Tac Toe is a simple two-player game where the players take turns marking a 3x3 grid with their designated symbols (usually "X" and "O") with the goal of getting three of their symbols in a row — either horizontally, vertically, or diagonally. Here's a step-by-step guide on how to play Tic Tac Toe:

Game Setup:

1. **Board Setup:**
   - The game is played on a 3x3 grid.
   - Each cell in the grid represents a position where a player can place their symbol.
2. **Player Assignment:**
   - There are two players, often referred to as "Player 1" and "Player 2."

- Player 1 typically uses " X, " and Player 2 uses " O ."

Gameplay :

3. Starting the Game :

- The game starts with an empty board .

4. Taking Turns :

- Players take turns to make a move .
- Player 1 (X) goes first, followed by Player 2 (O) , and they continue alternating turns .

5. Making a Move :

- On a player's turn, they choose an empty cell on the grid to place their symbol .
- Click on the chosen cell to mark it with the player's symbol .

6. Winning the Game :

- The game is won when a player successfully places three of their symbols in a row .
- A row can be horizontal, vertical, or diagonal .

7. End of the Game :

- If a player wins, the game ends, and the winning player is announced .
- If the board is filled with symbols, and there is no winner, the game is a draw .

8. Starting a New Game :

- After the game ends（either in a win or a draw）, players can start a new game.
- Some implementations include a "Reset" button to clear the board and start over.

Example Gameplay:

- Player 1（X）makes a move by clicking on an empty cell.
- Player 2（O）takes their turn, selecting another empty cell.
- Players continue taking turns until one player gets three symbols in a row, or the board is filled.

Winning Combinations:

- **Horizontal:**

mathematica Code

```
X | X | X
O | O |
  |   |
```

- **Vertical:**

mathematica Code

```
X | O |
X | O |
X |   |
```

- **Diagonal :**

mathematica Code

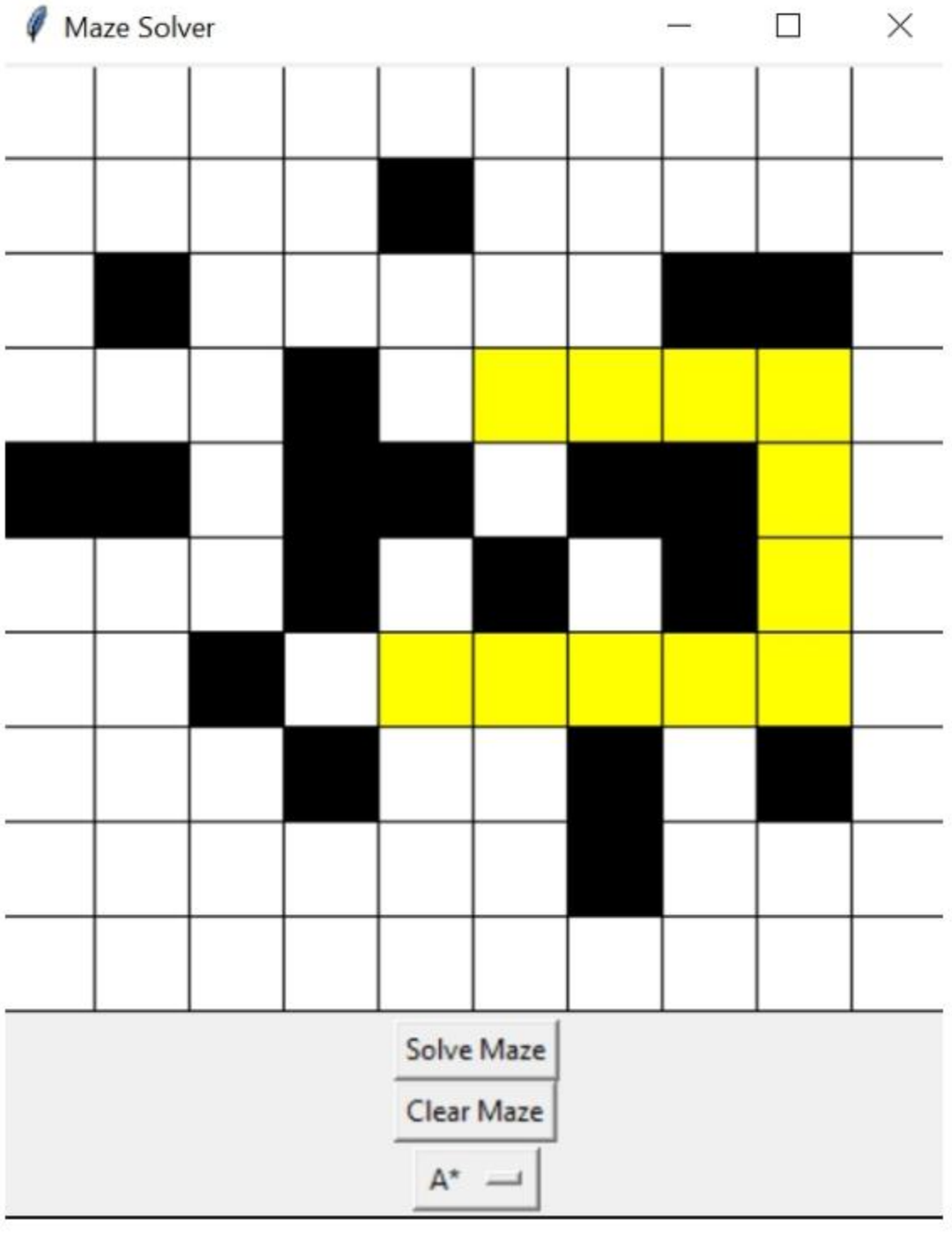
```
X | O |
  | X | O
  |   | X
```

Tips :

- Pay attention to the opponent's moves and plan ahead to block potential winning combinations .
- Try to create your own winning opportunities while preventing your opponent from doing the same .

Tic Tac Toe is a game of strategy and anticipation, often enjoyed for its simplicity and quick gameplay . It's a great introductory game for those new to strategy board games .

5. Maze Solver Game



```
import tkinter as tk
from queue import PriorityQueue, Queue
```

```
class MazeSolver:
  def __init__(self, root, rows, cols):
    self.root = root
    self.rows = rows
    self.cols = cols
    self.canvas_size = 400
    self.cell_size = self.canvas_size // max(rows, cols)
    self.canvas = tk.Canvas(
      root, width=self.canvas_size, height=self.canvas_size)
    self.canvas.pack()
    self.start = None
    self.end = None
    self.maze = [[0] * cols for _ in range(rows)]
    self.draw_grid()
    self.canvas.bind("<Button-1>", self.on_click)
    self.solve_button = tk.Button(
      root, text="Solve Maze", command=self.solve_maze)
    self.solve_button.pack()
    self.clear_button = tk.Button(
      root, text="Clear Maze", command=self.clear_maze)
    self.clear_button.pack()
```

```
    self.algorithm_var = tk.StringVar(root)
    self.algorithm_var.set("A*")
    self.algorithm_menu = tk.OptionMenu(
        root, self.algorithm_var, "A*", "Dijkstra", "BFS")
    self.algorithm_menu.pack()

def draw_grid(self):
    for i in range(self.rows):
        for j in range(self.cols):
            x1, y1 = j * self.cell_size, i * self.cell_size
            x2, y2 = x1 + self.cell_size, y1 + self.cell_size
            self.canvas.create_rectangle(
                x1, y1, x2, y2, fill="white", outline="black")

def on_click(self, event):
    col = event.x // self.cell_size
    row = event.y // self.cell_size
    if not self.start:
        self.start = (row, col)
        self.canvas.create_rectangle(col * self.cell_size, row * self.cell_size,
                                     (col + 1) * self.cell_size, (row + 1) * self.cell_size, fill="green")
    elif not self.end:
        self.end = (row, col)
        self.canvas.create_rectangle(col * self.cell_size, row * self.cell_size,
```

```
                    (col + 1) * self.cell_size, (row + 1) * self.cell_size, fill="red")
        else:
            self.toggle_obstacle(row, col)

    def toggle_obstacle(self, row, col):
        if self.maze[row][col] == 0:
            self.maze[row][col] = 1
            self.canvas.create_rectangle(col * self.cell_size, row * self.cell_size,
                                         (col + 1) * self.cell_size, (row + 1) * self.cell_size, fill="black")
        else:
            self.maze[row][col] = 0
            self.canvas.create_rectangle(col * self.cell_size, row * self.cell_size,
                                         (col + 1) * self.cell_size, (row + 1) * self.cell_size, fill="white")

    def clear_maze(self):
        self.start = None
        self.end = None
        self.maze = [[0] * self.cols for _ in range(self.rows)]
        self.canvas.delete("all")
        self.draw_grid()

    def solve_maze(self):
        algorithm = self.algorithm_var.get()
        path = self.run_algorithm(algorithm)
```

```
    if path:
        self.highlight_path(path)

def run_algorithm(self, algorithm):
    if algorithm == "A*":
        return self.astar()
    elif algorithm == "Dijkstra":
        return self.dijkstra()
    elif algorithm == "BFS":
        return self.bfs()
    else:
        raise ValueError("Unsupported algorithm")

def astar(self):
    start = self.start
    end = self.end
    open_set = PriorityQueue()
    open_set.put((0, start))
    came_from = {start: None}
    g_score = {start: 0}

    while not open_set.empty():
        current = open_set.get()[1]
        if current == end:
```

```
        path = []
        while current in came_from:
            path.append(current)
            current = came_from[current]
        return path[::-1]

    for neighbor in self.get_neighbors(current):
        tentative_g_score = g_score[current] + 1
        if tentative_g_score < g_score.get(neighbor, float('inf')):
            g_score[neighbor] = tentative_g_score
            f_score = tentative_g_score + self.heuristic(neighbor, end)
            open_set.put((f_score, neighbor))
            came_from[neighbor] = current

  return None

def dijkstra(self):
  start = self.start
  end = self.end
  open_set = PriorityQueue()
  open_set.put((0, start))
  came_from = {start: None}
  g_score = {start: 0}
```

```
    while not open_set.empty():
        current = open_set.get()[1]
        if current == end:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            return path[::-1]

        for neighbor in self.get_neighbors(current):
            tentative_g_score = g_score[current] + 1
            if tentative_g_score < g_score.get(neighbor, float('inf')):
                g_score[neighbor] = tentative_g_score
                open_set.put((tentative_g_score, neighbor))
                came_from[neighbor] = current

    return None

def bfs(self):
    start = self.start
    end = self.end
    queue = Queue()
    queue.put(start)
    came_from = {start: None}
```

```
  while not queue.empty():
    current = queue.get()
    if current == end:
      path = []
      while current in came_from:
        path.append(current)
        current = came_from[current]
      return path[::-1]

    for neighbor in self.get_neighbors(current):
      if neighbor not in came_from:
        queue.put(neighbor)
        came_from[neighbor] = current

  return None

def get_neighbors(self, cell):
  row, col = cell
  neighbors = []
  if row > 0 and self.maze[row - 1][col] == 0:
    neighbors.append((row - 1, col))
  if row < self.rows - 1 and self.maze[row + 1][col] == 0:
    neighbors.append((row + 1, col))
```

```
    if col > 0 and self.maze[row][col - 1] == 0:
        neighbors.append((row, col - 1))
    if col < self.cols - 1 and self.maze[row][col + 1] == 0:
        neighbors.append((row, col + 1))
    return neighbors

def heuristic(self, a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def highlight_path(self, path):
    for cell in path:
        row, col = cell
        self.canvas.create_rectangle(col * self.cell_size, row * self.cell_size,
                                     (col + 1) * self.cell_size, (row + 1) * self.cell_size, fill="yellow")
```

```
if __name__ == "__main__":
    root = tk.Tk()
    root.title("Maze Solver")
    maze_solver = MazeSolver(root, rows=10, cols=10)
    root.mainloop()
```

This Python script uses the Tkinter library to create a simple graphical user interface (GUI) for solving mazes . The maze - solving algorithms implemented include A* , Dijkstra's algorithm, and Breadth - First Search (BFS).

Here's a detailed breakdown of the script :

1. **Imports :**
    - tkinter : Used for creating the GUI .
    - Queue and PriorityQueue : Used for implementing the queue data structure for BFS and the priority queue for A* and Dijkstra's algorithm .
2. **MazeSolver Class :**
    - **Initialization (__init__):**
        - Initializes the GUI components, such as the main window (root) , canvas for drawing the maze, buttons for solving and clearing the maze, and an option menu for selecting the algorithm .
        - Sets up variables for the number of rows and columns in the maze, cell size, and other parameters .
        - Initializes the maze grid, start and end points, and binds the left - click event to the on_click method .
    - **Drawing the Grid (draw_grid):**

- Draws the initial grid on the canvas, where each cell is a rectangle with a white fill and black outline .

- **Handling Mouse Clicks (on_click):**
  - Determines the row and column of the clicked cell based on the mouse click coordinates .
  - Handles the placement of the start and end points and toggles obstacles when the mouse is clicked .
- **Toggling Obstacles (toggle_obstacle):**
  - Toggles between obstacle (black) and empty (white) cells when clicking on a cell .
- **Clearing the Maze (clear_maze):**
  - Resets the start and end points and clears the maze grid on the canvas .
- **Solving the Maze (solve_maze):**
  - Gets the selected algorithm from the option menu and runs the corresponding maze - solving algorithm .
  - Highlights the solution path on the canvas .
- **Running the Algorithm (run_algorithm):**
  - Chooses the appropriate algorithm based on the user's selection .

- *A Algorithm (astar), Dijkstra's Algorithm (dijkstra), and BFS Algorithm (bfs):**
    - Each algorithm uses a different approach to explore the maze and find the solution path.
    - A* and Dijkstra's algorithms use priority queues, while BFS uses a regular queue.
- **Getting Neighbors (get_neighbors):**
    - Returns a list of neighboring cells that are accessible from the current cell.
- **Heuristic Function (heuristic):**
    - Calculates a simple heuristic value (Manhattan distance) between two cells.
- **Highlighting the Solution Path (highlight_path):**
    - Draws a yellow rectangle for each cell in the solution path on the canvas.

3. **Main Block (__main__):**
    - Creates a Tkinter window (root), sets its title, and initializes an instance of the MazeSolver class with a 10x10 maze.
    - Starts the Tkinter main event loop.

In summary, the script provides a basic interactive maze - solving application with a GUI, allowing users to create and solve mazes using different algorithms .

How To Play Maze Solver Game

The provided Python script creates a simple maze - solving application with a graphical user interface (GUI). While it's not explicitly designed as a game, you can follow these steps to interact with the Maze Solver application :

1. **Run the Script :**
    - Save the provided Python script to a file, for example, maze_solver . py .
    - Open a terminal or command prompt and navigate to the directory containing the script .
    - Run the script using the command : python maze_solver . py (or the equivalent command for your Python environment).
2. **GUI Interface :**
    - A window titled " Maze Solver " will appear .
    - The GUI consists of a canvas where you can create a maze, buttons for solving and clearing the maze, and an option menu for selecting the solving algorithm .

3. **Creating a Maze :**

- Left - click on the white cells in the canvas to create a maze . Clicking on a cell toggles it between an empty cell and an obstacle (black cell).
- To set the start and end points :
  - Click on a white cell to set the start point (green rectangle).
  - Click on another white cell to set the end point (red rectangle).

4. **Selecting an Algorithm :**

- Choose the solving algorithm from the drop - down menu (A * , Dijkstra, or BFS).

5. **Solving the Maze :**

- Click the " Solve Maze " button to apply the selected algorithm and find the solution path .

6. **Viewing the Solution :**

- The solution path will be highlighted in yellow on the canvas .

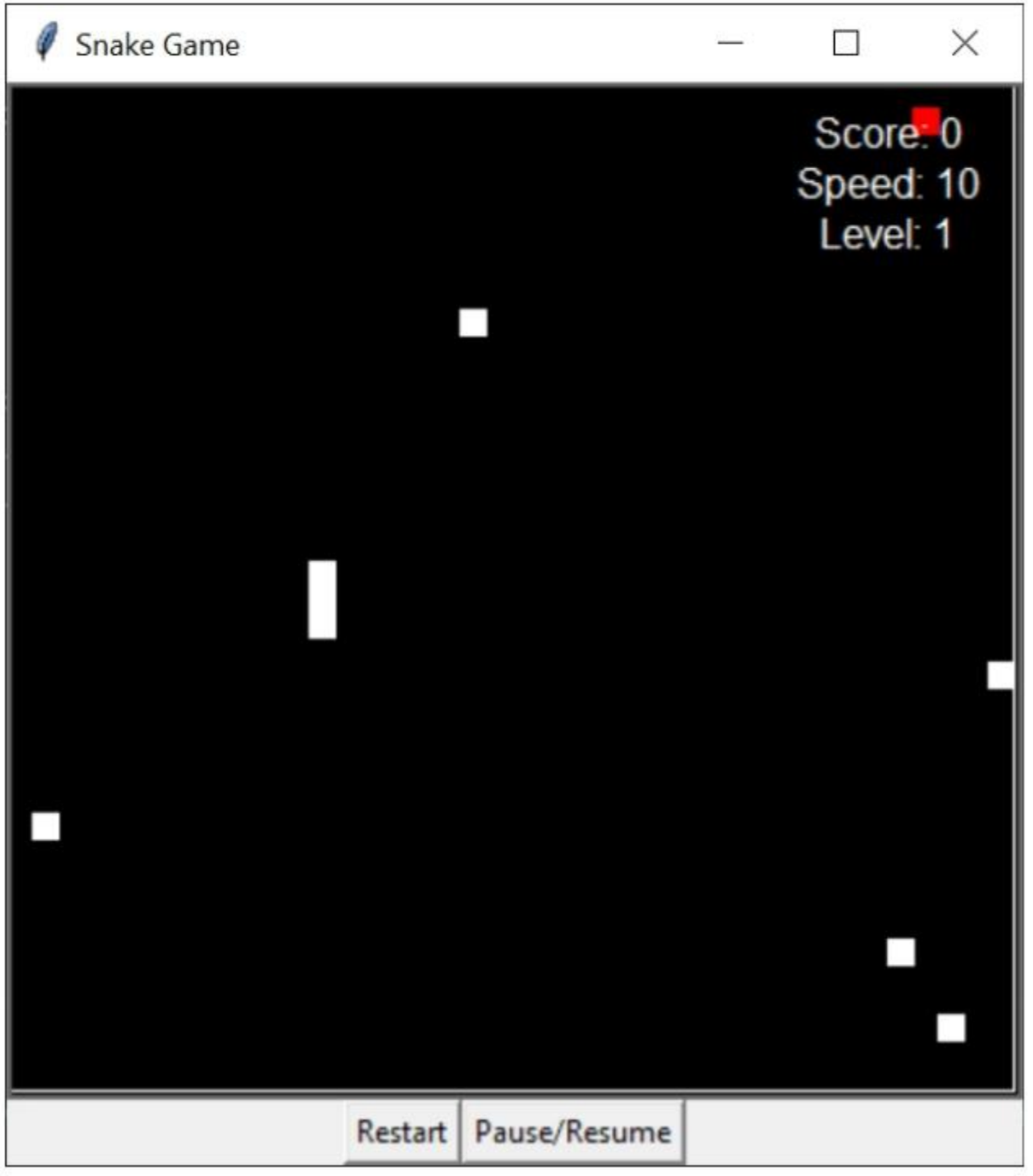
7. **Clearing the Maze :**

- Click the " Clear Maze " button to reset the maze, removing obstacles, start, and end points .

8. **Repeat and Experiment :**

- You can create different mazes, change the start and end points, and explore how different algorithms find solutions .

6. Snake Game



```
import tkinter as tk
import random
import winsound  # For Windows systems
```

```
class SnakeGame:
  def __init__(self, master, width=400, height=400):
    self.master = master
    self.master.title("Snake Game")
    self.canvas = tk.Canvas(self.master, width=width,
                            height=height, bg="black")
    self.canvas.pack()

    self.canvas.focus_set()

    self.snake = [(100, 100), (90, 100), (80, 100)]
    self.direction = "Right"
    self.food = self.create_food()
    self.level = 1  # Initialize level before calling create_obstacles
    self.obstacles = self.create_obstacles()
    self.score = 0
    self.high_score = 0
    self.speed = 100
    self.game_over_flag = False
    self.paused = False
```

```
    self.score_display = self.canvas.create_text(
        350, 20, text="Score: 0", fill="white", font=("Helvetica", 12))
    self.speed_display = self.canvas.create_text(
        350, 40, text="Speed: 1\nLevel: 1", fill="white", font=("Helvetica", 12))

    self.button_frame = tk.Frame(self.master)
    self.button_frame.pack(side="bottom")

    restart_button = tk.Button(
        self.button_frame, text="Restart", command=self.restart_game)
    restart_button.pack(side="left")

    pause_button = tk.Button(
        self.button_frame, text="Pause/Resume", command=self.toggle_pause)
    pause_button.pack(side="left")

    self.canvas.bind("<Key>", self.change_direction)
    self.master.after(self.speed, self.update)

    self.draw()  # Draw the initial state

def create_food(self):
    x = random.randint(1, 39) * 10
```

```
    y = random.randint(1, 39) * 10
    self.canvas.create_rectangle(
        x, y, x + 10, y + 10, outline="red", fill="red", tags="food")
    winsound.Beep(523, 100)  # Beep sound when the snake eats food
    return x, y

def create_obstacles(self):
    obstacles = []
    for _ in range(5 * self.level):  # Adjust obstacle count based on level
        x = random.randint(1, 39) * 10
        y = random.randint(1, 39) * 10
        self.canvas.create_rectangle(
            x, y, x + 10, y + 10, outline="white", fill="white", tags="obstacle")
        obstacles.append((x, y))
    return obstacles

def move(self):
    if self.game_over_flag or self.paused:
        return

    head = self.snake[0]
    if self.direction == "Right":
        new_head = (head[0] + 10, head[1])
    elif self.direction == "Left":
```

```
        new_head = (head[0] - 10, head[1])
    elif self.direction == "Up":
        new_head = (head[0], head[1] - 10)
    elif self.direction == "Down":
        new_head = (head[0], head[1] + 10)

    self.snake.insert(0, new_head)

    if new_head == self.food:
        self.score += 1
        self.canvas.delete("food")
        self.food = self.create_food()
        self.increase_speed()
        self.check_win()  # Check if the player has won after increasing the score
    else:
        self.canvas.delete(self.snake[-1])
        self.snake.pop()

    self.check_collision()

def increase_speed(self):
    level_thresholds = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]

    if self.score in level_thresholds and self.level < len(level_thresholds):
```

```
        self.level += 1
        self.create_obstacles()
        self.canvas.itemconfig(
          self.speed_display, text=f"Speed: {1000 // self.speed}\nLevel: {self.level}")
        winsound.PlaySound("level_up.wav", winsound.SND_FILENAME)

        if self.level == len(level_thresholds):
          self.display_completion_message()

    def display_completion_message(self):
      self.canvas.create_text(
        200, 200, text="All Levels Completed!\nCongratulations!", fill="white", font=("Helvetica", 16), tags="gameover")

    def check_collision(self):
      head = self.snake[0]
      if (
        head[0] < 0
        or head[0] >= 400
        or head[1] < 0
        or head[1] >= 400
        or head in self.snake[1:]
        or head in self.obstacles
      ):
        self.game_over()
```

```
def check_win(self):
  if self.level == 4 and self.score >= 20:
    self.game_over_flag = True
    if self.score > self.high_score:
      self.high_score = self.score
    self.canvas.create_text(
      200, 200, text=f"Congratulations!\nYou passed all levels!\nScore: {self.score}\nHigh Score: {self.high_score}",
      fill="white", font=("Helvetica", 16), tags="gameover")
    # Play a win sound
    winsound.PlaySound("game_win.wav", winsound.SND_FILENAME)

def game_over(self):
  self.game_over_flag = True
  if self.score > self.high_score:
    self.high_score = self.score
  self.canvas.create_text(
    200, 200, text=f"Game Over\nScore: {self.score}\nHigh Score: {self.high_score}",
    fill="white", font=("Helvetica", 16), tags="gameover")
  winsound.PlaySound("game_over.wav", winsound.SND_FILENAME)

def restart_game(self):
  self.canvas.delete("all")
  self.snake = [(100, 100), (90, 100), (80, 100)]
```

```
    self.direction = "Right"
    self.food = self.create_food()
    self.obstacles = self.create_obstacles()
    self.score = 0
    self.level = 1
    self.speed = 100
    self.game_over_flag = False
    self.paused = False

    self.draw()
    self.update()

def toggle_pause(self):
    self.paused = not self.paused

def update(self):
    self.move()
    self.draw()
    if not self.game_over_flag:
        self.master.after(self.speed, self.update)

def draw(self):
    self.canvas.delete("all")
    # Border around the game area
```

```
self.canvas.create_rectangle(0, 0, 400, 400, outline="white")

for segment in self.snake:
    self.canvas.create_rectangle(
        segment[0], segment[1], segment[0] + 10, segment[1] + 10, outline="white", fill="white")
self.canvas.create_rectangle(
    self.food[0], self.food[1], self.food[0] + 10, self.food[1] + 10, outline="red", fill="red")

for obstacle in self.obstacles:
    self.canvas.create_rectangle(
        obstacle[0], obstacle[1], obstacle[0] + 10, obstacle[1] + 10, outline="white", fill="white")

# Display the score
self.canvas.create_text(
    350, 20, text=f"Score: {self.score}", fill="white", font=("Helvetica", 12))

# Display the speed
self.canvas.create_text(
    350, 40, text=f"Speed: {1000 // self.speed}", fill="white", font=("Helvetica", 12))

# Display the level
self.canvas.create_text(
    350, 60, text=f"Level: {self.level}", fill="white", font=("Helvetica", 12))
```

```
  def change_direction(self, event):
    if event.keysym in ["Up", "Down", "Left", "Right"]:
      if (
        (event.keysym == "Up" and self.direction != "Down")
        or (event.keysym == "Down" and self.direction != "Up")
        or (event.keysym == "Left" and self.direction != "Right")
        or (event.keysym == "Right" and self.direction != "Left")
      ):
        self.direction = event.keysym
```

```
if __name__ == "__main__":
  root = tk.Tk()
  game = SnakeGame(root)
  root.mainloop()
```

This is a Snake Game implemented using the Tkinter library in Python . Let's break down the code and understand each component :

1. **Imports :**

Python Code

```
import tkinter as tk
import random
```

```
import winsound
```

- tkinter : The standard Python interface to the Tk GUI toolkit .
- random : Used for generating random numbers for the initial positions of the food and obstacles .
- winsound : For playing sound effects on Windows .

2. **SnakeGame Class :**

Python Code

```
class SnakeGame :
```

- This class represents the main game logic .

3. **Initialization :**

Python Code

```
def __init__ ( self, master, width = 400, height = 400 ):
```

- The constructor initializes the game with a Tkinter window (master) and sets the **def**ault width and height of the game area .
- It sets up the canvas, snake, initial direction, food, level, obstacles, score, speed, and other game - related attributes .
- The game state is displayed on the Tkinter window using labels and buttons .

4. **create_food method :**

Python Code

```
def create_food ( self ):
```

- Generates random coordinates for the food and creates a red rectangle at that position on the canvas .
- Plays a beep sound when the snake eats the food .

5. **create_obstacles method :**

Python Code

```
def create_obstacles ( self ):
```

- Generates a specified number of obstacles at random positions on the canvas .
- Obstacles are white rectangles .

6. **move method :**

Python Code

```
def move ( self ):
```

- Updates the snake's position based on the current direction .
- Handles collisions with food, updates the score, and increases the speed and level when certain conditions are met .

- Checks for collisions with obstacles or the game boundaries .

7. **increase_speed method :**

Python Code

```
def increase_speed ( self ):
```

- Increases the game speed based on the player's score and updates the level .

8. **display_completion_message method :**

Python Code

```
def display_completion_message ( self ):
```

- Displays a congratulatory message when the player completes all levels .

9. **check_collision method :**

Python Code

```
def check_collision ( self ):
```

- Checks for collisions with walls, snake body, and obstacles .
- Calls the game_over method if a collision is detected .

10. **check_win method :**

Python Code

```
def check_win ( self ):
```

- Checks if the player has won the game (reached a certain level and score).
- Displays a victory message and plays a winning sound .

11. **game_over method :**

Python Code

```
def game_over ( self ):
```

- Displays a game over message with the final score and high score .
- Plays a game over sound .

12. **restart_game method :**

Python Code

```
def restart_game ( self ):
```

- Resets the game state to its initial values and starts a new game .

13. **toggle_pause method :**

Python Code

```
def toggle_pause ( self ):
```

- Pauses or resumes the game when the pause button is clicked .

14. **update method :**

Python Code

```
def update ( self ):
```

- Updates the game state at regular intervals, allowing for continuous movement of the snake .
- Calls the move and draw methods .

15. **draw method :**

Python Code

```
def draw ( self ):
```

- Clears the canvas and redraws the game elements (snake, food, obstacles, score, speed, level).

16. **change_direction method :**

Python Code

```
def change_direction ( self, event ):
```

- Handles user input to change the direction of the snake .

17. **Main Execution :**

Python Code

```
if __name__ == " __main__ ":
    root = tk . Tk ()
    game = SnakeGame ( root )
    root . mainloop ()
```

- Creates a Tkinter window and starts the game loop .

In summary, this Snake Game is a complete implementation with features such as snake movement, collision detection, scoring, increasing difficulty levels, obstacles, and a graphical user interface using Tkinter . The game also includes sound effects for various events .

How To Play Snake Game

To play the Snake Game, follow these instructions :

1. **Start the Game :**

- Run the Python script containing the Snake Game code .
- A window will appear with the title " Snake Game ."

2. **Initial Setup :**

- The game starts with a snake (a series of white rectangles) and a red square representing food .
- The snake initially moves to the right .

3. **Control the Snake :**

- Use the arrow keys (Up, Down, Left, Right) to control the direction of the snake .
- The snake will continuously move in the chosen direction until the game ends .

4. **Objective :**

- The goal is to navigate the snake to eat the red food squares .
- Each time the snake consumes food, it grows longer, and the player earns points .

5. **Avoid Collisions :**

- Avoid running into the walls of the game area .
- Avoid colliding with the snake's own body .

- Avoid colliding with white obstacles that may appear on the screen .

6. **Scoring :**

- The score is displayed at the top of the game window .
- Each time the snake eats food, the score increases .

7. **Speed and Levels :**

- As the player accumulates points, the game speed increases .
- There are multiple levels in the game, and with each level, the difficulty increases .
- The current speed, level, and score are displayed on the right side of the game window .

8. **Winning :**

- If you reach a certain level and achieve a specific score, you win the game .
- A congratulatory message is displayed, and a victory sound plays .

9. **Losing :**

- The game ends if the snake collides with a wall, itself, or an obstacle .
- A game over message is displayed with your final score, and a game over sound plays .

10. **Restart or Pause :**

- You can restart the game by clicking the " Restart " button at the bottom .

- The " Pause / Resume " button allows you to pause and resume the game .

11. **Enjoy the Game :**

- Have fun playing the Snake Game and try to achieve the highest score possible !

Remember, the key to success is strategic movement, avoiding obstacles, and growing the snake by consuming food . As the game progresses, the challenge increases, making it an engaging and entertaining experience .

7. Memory Puzzle Game

```
import tkinter as tk
from tkinter import messagebox
import random
```

```
class MemoryPuzzle:
  def __init__(self, root, rows=6, columns=6):
    self.root = root
    self.root.title("Memory Puzzle")

    self.rows = rows
    self.columns = columns
    self.tiles = [i for i in range(1, (rows * columns) // 2 + 1)] * 2
    random.shuffle(self.tiles)

    self.buttons = []
    self.create_buttons()

    self.first_click = None
    self.moves = 0

    # Set the initial form width and height
    self.initial_width = 350
    self.initial_height = 350
    self.center_window()
```

```
def create_buttons(self):
  for i in range(self.rows):
    for j in range(self.columns):
      index = i * self.columns + j
      button = tk.Button(self.root, text=" ", width=5, height=2,
                command=lambda idx=index: self.flip_tile(idx))
      button.grid(row=i, column=j, padx=5, pady=5)
      self.buttons.append(button)

  # Add "Play Again" button
  play_again_button = tk.Button(
    self.root, text="Play Again", command=self.reset_game)
  play_again_button.grid(row=self.rows, column=self.columns // 2)

def flip_tile(self, index):
  if self.buttons[index]["state"] == tk.NORMAL:
    self.buttons[index].config(
      text=str(self.tiles[index]), state=tk.DISABLED)
    if self.first_click is None:
      self.first_click = index
    else:
      self.moves += 1
      self.root.after(
```

```
            1000, lambda idx=index, first_click=self.first_click: self.check_match(idx, first_click))
        self.first_click = None

def check_match(self, index, first_click):
    if self.tiles[first_click] == self.tiles[index]:
        messagebox.showinfo("Match", "You found a match!")
        self.buttons[first_click].config(state=tk.DISABLED)
        self.buttons[index].config(state=tk.DISABLED)
    else:
        self.buttons[first_click].config(text=" ", state=tk.NORMAL)
        self.buttons[index].config(text=" ", state=tk.NORMAL)

    if all(self.buttons[i]["state"] == tk.DISABLED for i in range(self.rows * self.columns)):
        self.show_game_over_message()

def reset_game(self):
    # Reset the game by destroying the current window and creating a new one
    self.root.destroy()
    new_root = tk.Tk()
    new_game = MemoryPuzzle(new_root, rows=6, columns=6)
    new_root.mainloop()

def show_game_over_message(self):
    messagebox.showinfo(
```

```
        "Game Over", f"Congratulations! You won in {self.moves} moves.")

    def center_window(self):
        # Calculate the center position on the screen
        screen_width = self.root.winfo_screenwidth()
        screen_height = self.root.winfo_screenheight()
        x_position = (screen_width - self.initial_width) // 2
        y_position = (screen_height - self.initial_height) // 2

        self.root.geometry(
            f"{self.initial_width}x{self.initial_height}+{x_position}+{y_position}")
```

```
if __name__ == "__main__":
    root = tk.Tk()
    game = MemoryPuzzle(root, rows=6, columns=6)
    root.mainloop()
```

This Python script uses the Tkinter library to create a simple memory puzzle game . Let's go through the code step by step :

1. **Importing Libraries :**

Python Code

```
import tkinter as tk
```

```
from tkinter import messagebox
import random
```

- tkinter : GUI library used for creating the graphical interface .
- messagebox : Part of Tkinter for displaying dialog boxes .
- random : Used for shuffling the tiles in the memory puzzle .

2. **MemoryPuzzle Class :**

Python Code

```
class MemoryPuzzle :
    def __init__ ( self, root, rows = 6, columns = 6 ):
        # Initialization method for the MemoryPuzzle class .
        # Takes the Tkinter root window and optional rows and columns parameters .

        # Initialize the Tkinter root window .
        self . root  =  root
        self . root . title (" Memory Puzzle ")

        # Set the number of rows and columns in the game grid .
        self . rows  =  rows
```

```
        self . columns  =  columns

        # Create a list of tile values and shuffle it .
        self . tiles  = [ i for i in range ( 1, ( rows  *  columns ) //  2 +  1 )] *  2
        random . shuffle ( self . tiles )

        # Initialize lists to store buttons and other variables .
        self . buttons  = []
        self . first_click  =  None
        self . moves  =  0

        # Set the initial form width and height, and center the window .
        self . initial_width  =  350
        self . initial_height  =  350
        self . center_window ()

        # Create buttons and the  " Play Again "  button .
        self . create_buttons ()
```

Other methods (create_buttons, flip_tile, check_match, reset_game, show_game_over_message, center_window) are **def**ined within the class .

- __init__ : Initializes the MemoryPuzzle object . Sets up the game grid, buttons, and other parameters .
- create_buttons : Creates buttons for the game grid and the " Play Again " button .
- flip_tile : Handles the click event for each tile, flipping it and checking for matches .
- check_match : Compares the values of two clicked tiles and updates the game accordingly .
- reset_game : Destroys the current window and creates a new one to reset the game .
- show_game_over_message : Displays a message box when the game is completed .
- center_window : Centers the Tkinter window on the screen .

3. **Main Section :**

Python Code

```
if __name__ == " __main__ ":
    root = tk . Tk ()
    game = MemoryPuzzle ( root, rows = 6, columns = 6 )
    root . mainloop ()
```

- Checks if the script is being run as the main module .
- Creates the Tkinter root window and initializes the MemoryPuzzle game .
- Starts the Tkinter main event loop .

4. **Execution :**

- The script creates a Tkinter window with a 6x6 grid of buttons, representing the memory puzzle .
- Each button has a hidden value that is revealed when clicked .
- The goal is to find matching pairs by clicking on two buttons with the same value .
- The game provides feedback on successful matches and displays a " Game Over " message when all pairs are found .

Note : The game window is initially centered on the screen, and the " Play Again " button allows the player to restart the game after completing it .

How To Play Memory Puzzle Game

The Memory Puzzle game is a classic memory matching game where the player needs to find matching pairs of tiles . Here's a step - by - step guide on how to play the Memory Puzzle game :

1. **Objective :**
    - The goal of the game is to match all pairs of tiles .
2. **Game Setup :**
    - When you start the game, a grid of face - down tiles is displayed on the screen .
    - Each tile has a hidden value .
3. **Game Mechanics :**
    - Click on a tile to reveal its value .
    - Then, click on another tile to reveal its value .
    - If the values of the two revealed tiles match, they stay face - up, and you score a point .
    - If the values do not match, the tiles are flipped face - down again .
4. **Remembering the Tiles :**
    - Pay attention to the values of the tiles when they are revealed .

- Try to remember the locations of matching pairs .

5. **Strategy :**

- The key to success is to remember the positions of the tiles and match them efficiently .
- Use your memory to recall where different values are located in the grid .

6. **Game Progress :**

- The game keeps track of the number of moves you make .
- Try to complete the game with the fewest moves possible .

7. **Winning the Game :**

- Continue revealing and matching pairs until all tiles are face - up .
- Once all pairs are matched, a " Game Over " message will be displayed, showing the number of moves it took to complete the game .

8. **Restarting the Game :**

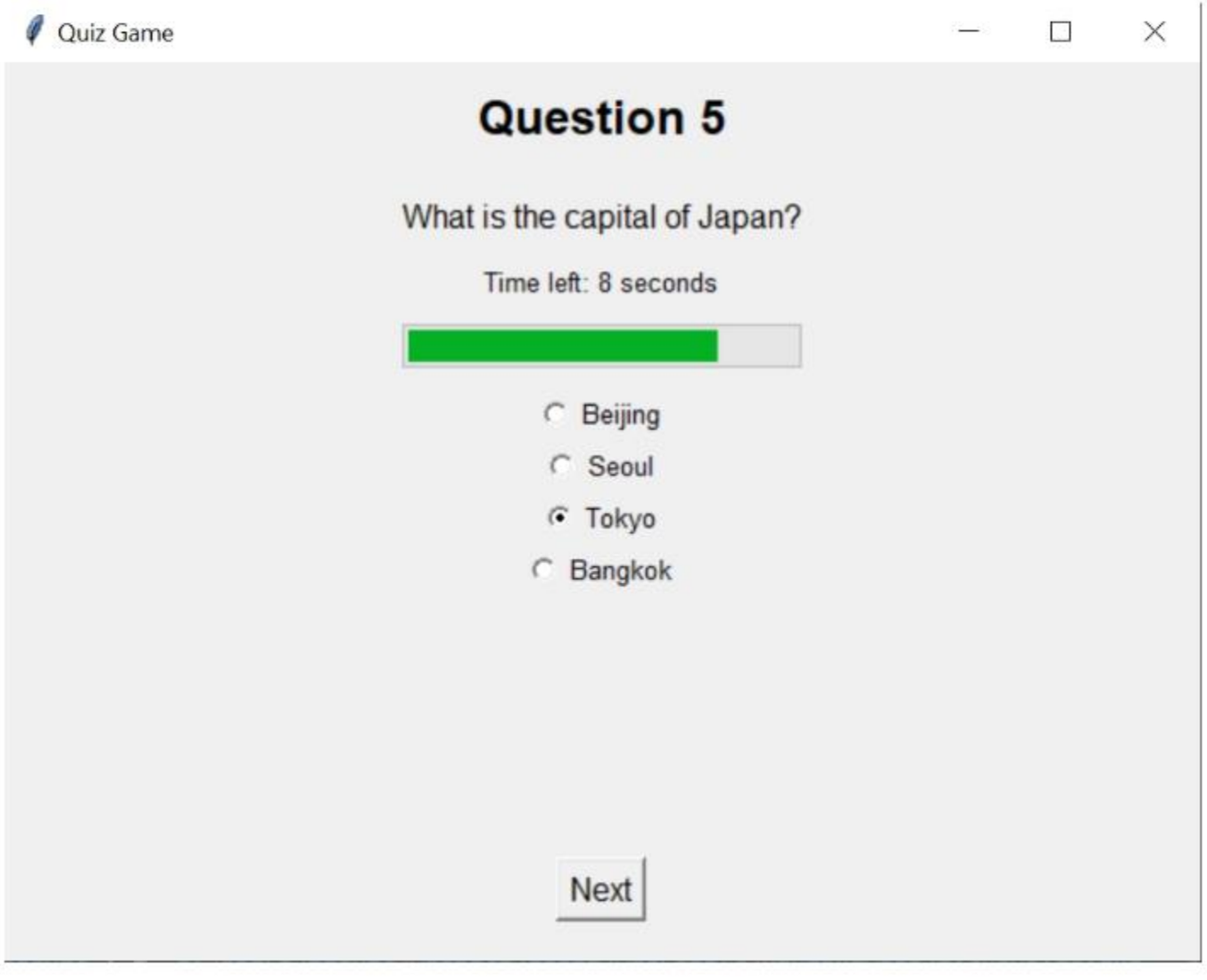
- If you want to play again, you can click the " Play Again " button at the bottom of the game window .
- This will reset the game, shuffle the tiles, and allow you to start a new game .

9. **Have Fun :**

- Enjoy the process of testing and improving your memory skills .

Remember, the Memory Puzzle game is not only entertaining but also a great exercise for your memory and concentration . Good luck, and have fun playing !

8. Quiz Game



```
import tkinter as tk
from tkinter import messagebox
from tkinter import ttk
import time
```

```
class QuizGame:
  def __init__(self, root):
    self.root = root
    self.root.title("Quiz Game")
    self.root.geometry("600x450")
    self.root.attributes('-topmost', True)

    self.current_question = 0
    self.score = 0
    self.timer_seconds = 10
    self.timer_label = None
    self.progress_bar = None

    self.questions = [
      {
        "question": "What is the capital of France?",
        "options": ["Berlin", "Madrid", "Paris", "Rome"],
        "correct_answer": "Paris"
      },
```

```
    {
        "question": "Which planet is known as the Red Planet?",
        "options": ["Mars", "Venus", "Jupiter", "Saturn"],
        "correct_answer": "Mars"
    },
    {
        "question": "What is the largest mammal in the world?",
        "options": ["Elephant", "Blue Whale", "Giraffe", "Hippopotamus"],
        "correct_answer": "Blue Whale"
    },
    {
        "question": "Which programming language is this quiz written in?",
        "options": ["Python", "Java", "C++", "JavaScript"],
        "correct_answer": "Python"
    },
    {
        "question": "What is the capital of Japan?",
        "options": ["Beijing", "Seoul", "Tokyo", "Bangkok"],
        "correct_answer": "Tokyo"
    }
]

self.create_widgets()
```

```
def center_window(self):
    self.root.update_idletasks()
    window_width = self.root.winfo_width()
    window_height = self.root.winfo_height()
    position_left = int(
        self.root.winfo_screenwidth() / 2 - window_width / 2)
    position_top = int(self.root.winfo_screenheight() /
                       2 - window_height / 2)
    self.root.geometry(f"+{position_left}+{position_top}")

def create_widgets(self):
    self.title_label = tk.Label(
        self.root, text="Quiz Game", font=("Helvetica", 18, "bold"))
    self.title_label.pack(pady=10)

    self.label_question = tk.Label(
        self.root, text="", font=("Helvetica", 12))
    self.label_question.pack(pady=10)

    self.var_option = tk.StringVar()
    self.option_buttons = []
    for option in self.questions[self.current_question]["options"]:
        radio_button = tk.Radiobutton(
            self.root, text=option, variable=self.var_option, value=option, font=("Helvetica", 10))
```

```
        self.option_buttons.append(radio_button)
        radio_button.pack()

    self.btn_next = tk.Button(
        self.root, text="Next", command=self.next_question, font=("Helvetica", 12))
    self.btn_next.pack(side=tk.BOTTOM, pady=20)

    self.timer_label = tk.Label(
        self.root, text=f"Time left: {self.timer_seconds} seconds", font=("Helvetica", 10))
    self.timer_label.pack()

    self.progress_bar = ttk.Progressbar(
        self.root, length=200, mode="determinate")
    self.progress_bar.pack(pady=10)

    self.update_question()
    self.start_timer()

def next_question(self):
    user_answer = self.var_option.get()

    if user_answer == self.questions[self.current_question]["correct_answer"]:
        self.score += 1
```

```
    self.current_question += 1

    if self.current_question < len(self.questions):
        self.update_question()
    else:
        self.show_result()

def update_question(self):
    self.title_label.config(text=f"Question {self.current_question + 1}")
    self.label_question.config(
        text=self.questions[self.current_question]["question"])

    self.var_option.set(None)
    for button in self.option_buttons:
        button.destroy()

    self.option_buttons = []
    for option in self.questions[self.current_question]["options"]:
        radio_button = tk.Radiobutton(
            self.root, text=option, variable=self.var_option, value=option, font=("Helvetica", 10))
        self.option_buttons.append(radio_button)
        radio_button.pack()

    self.timer_seconds = 10  # Reset the timer for each question
```

```
    self.start_timer()

def start_timer(self):
    self.timer_label.config(text=f"Time left: {self.timer_seconds} seconds")
    self.progress_bar["value"] = 100  # Reset progress bar
    self.update_timer()

def update_timer(self):
    if self.current_question < len(self.questions):
        self.timer_seconds -= 1
        self.timer_label.config(text=f"Time left: {self.timer_seconds} seconds")
        self.progress_bar["value"] = (self.timer_seconds / 10) * 100

        if self.timer_seconds >= 0:
            # Increase the delay to slow down the progress bar
            self.root.after(6000, self.update_timer)
        else:
            self.next_question()
```

```
def show_result(self):
    result_text = f"You scored {self.score} out of {len(self.questions)}!"
    messagebox.showinfo("Quiz Completed", result_text)
```

```
    #self.root.destroy()
```

```
if __name__ == "__main__":
    root = tk.Tk()
    app = QuizGame(root)
    app.center_window()
    root.mainloop()
```

This is a simple quiz game implemented using the Tkinter library in Python . Let's break down the code and understand each part in detail :

1. **Importing Libraries :**

Python Code

```
import tkinter as tk
from tkinter import messagebox
from tkinter import ttk
import time
```

- tkinter : This is the standard GUI (Graphical User Interface) toolkit that comes with Python .
- messagebox : Provides a set of convenience functions for creating standard modal dialogs .

- ttk : Themed Tkinter, which provides access to the Tk themed widget set .
- time : Used for handling time - related functionality .

2. **QuizGame Class :**

Python Code

```
class QuizGame :
```

- This class encapsulates the entire quiz game .

3. **Initializer (__init__):**

Python Code

```
def __init__ ( self, root ):
```

- Initializes the QuizGame class with the given root Tkinter window .

4. **Window Configuration :**

- Sets up the basic configuration for the Tkinter window, including title, size, and position .
- Initializes variables like current_question , score , timer_seconds , timer_label , and progress_bar .

5. **List of Questions :**

Python Code

```
self . questions  = [...]
```

- Contains a list of dictionaries, where each dictionary represents a question with its options and correct answer .

6. create_widgets **Method :**

- Configures and creates various widgets (GUI components) such as labels, radio buttons, buttons, timer label, and progress bar .
- Calls update_question and start_timer to initialize the first question and start the timer .

7. center_window **Method :**

- Centers the Tkinter window on the screen .

8. next_question **Method :**

- Handles the logic for moving to the next question .
- Checks if the user's answer is correct, updates the score, and moves to the next question .
- If there are no more questions, it calls the show_result method .

9. update_question **Method :**

- Updates the GUI with the current question and its options .
- Resets the timer for each question .
- Calls start_timer to initiate the timer countdown .

10. start_timer **Method :**
    - Initializes and starts the timer, updating the timer label and progress bar .
11. update_timer **Method :**
    - Updates the timer label and progress bar based on the remaining time .
    - Calls itself recursively with a delay using after method until the timer reaches zero .
12. show_result **Method :**
    - Displays a message box with the quiz result, showing the user's score .
13. **Main Block (__main__):**
    - Creates a Tkinter root window and initializes the QuizGame class .
    - Centers the window and starts the Tkinter event loop .

This implementation provides a simple interactive quiz game with a countdown timer for each question and a final score display at the end . Players can answer questions by selecting options, and the game keeps track of their score .

How To Play Quiz Game

To play the quiz game, follow these steps :

1. **Run the Python Script :**

   - Save the provided Python script with the quiz game code in a file (e.g., quiz_game . py).
   - Open a terminal or command prompt .
   - Navigate to the directory containing the script .
   - Run the script by executing the command :

```
Code
python quiz_game . py
```

2. This will launch the quiz game window .
3. **Answering Questions :**

   - The quiz game window will display the first question and multiple - choice options .
   - Click on the radio button next to the answer you think is correct .

4. **Next Question :**

   - Click the " Next " button to move to the next question .
   - The timer will reset for each question .

5. **Timer Countdown :**

- Pay attention to the timer label and progress bar .
- You have a limited time to answer each question (**def**ault is 10 seconds).

6. **Scoring :**

- Your score increases when you select the correct answer .
- The total score is displayed at the end of the quiz .

7. **End of Quiz :**

- After answering all the questions, a message box will appear with your final score .

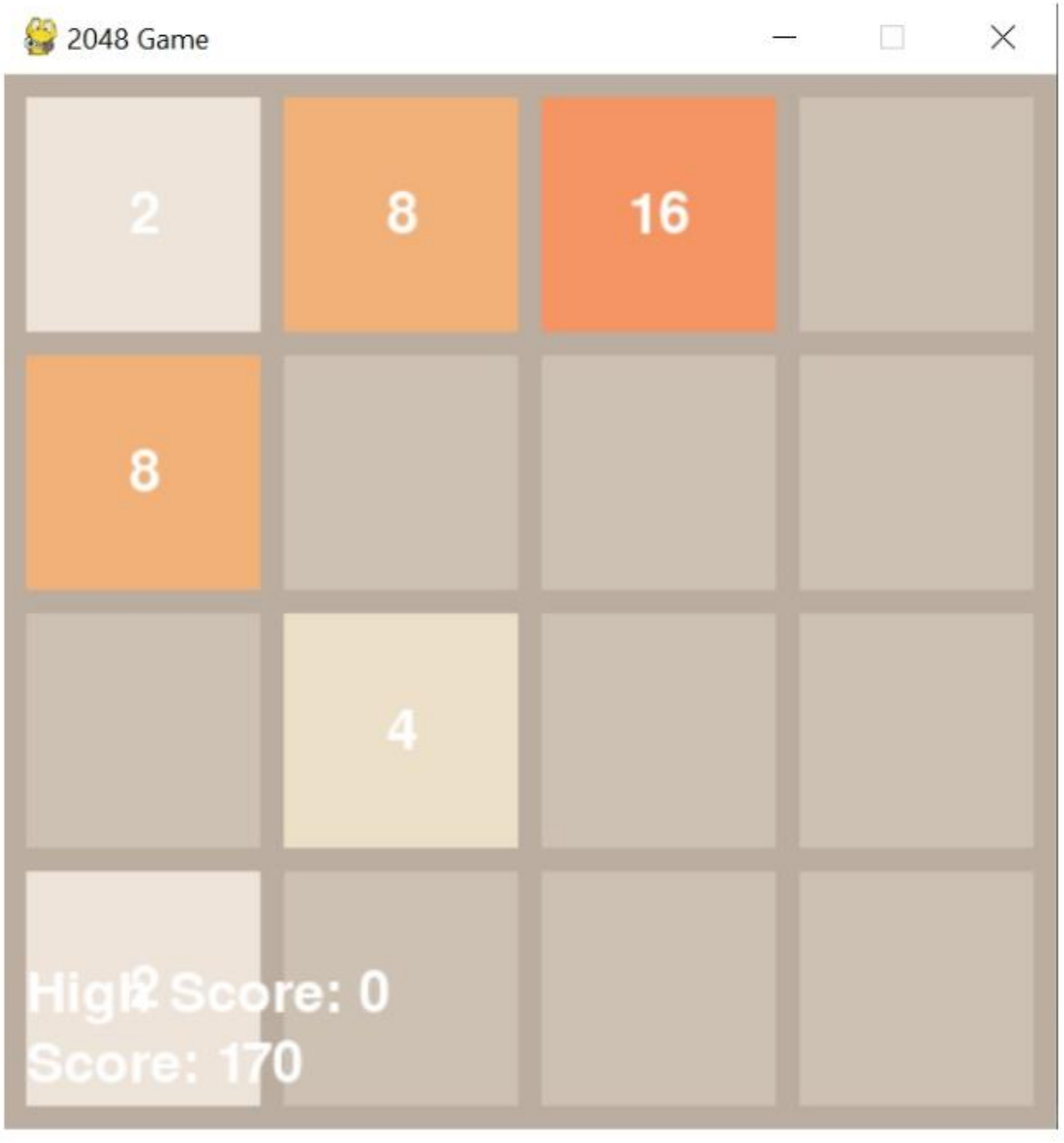
8. **Close the Game :**

- You can close the quiz game window after viewing your score .

9. **Note :**

- The quiz game has a set of pre**def**ined questions and answers . If you want to customize the quiz content, you can modify the `self . questions` list in the script .

9. 2048 Game



```
import pygame
import random

# Initialize pygame
pygame.init()

# Constants
GRID_SIZE = 4
TILE_SIZE = 100
GRID_MARGIN = 10
SCREEN_SIZE = (GRID_SIZE * TILE_SIZE + (GRID_SIZE + 1) * GRID_MARGIN,
               GRID_SIZE * TILE_SIZE + (GRID_SIZE + 1) * GRID_MARGIN)
BACKGROUND_COLOR = (187, 173, 160)
GRID_COLOR = (205, 193, 180)
FONT_COLOR = (255, 255, 255)
```

```
# Initialize screen
screen = pygame.display.set_mode(SCREEN_SIZE)
pygame.display.set_caption("2048 Game")

# Fonts
font = pygame.font.Font(None, 36)

# Colors for each tile value
```

```
TILE_COLORS = {
  0: (205, 193, 180),
  2: (238, 228, 218),
  4: (237, 224, 200),
  8: (242, 177, 121),
  16: (245, 149, 99),
  32: (246, 124, 95),
  64: (246, 94, 59),
  128: (237, 207, 114),
  256: (237, 204, 97),
  512: (237, 200, 80),
  1024: (237, 197, 63),
  2048: (237, 194, 46),
}

# Constants for additional features
ANIMATION_SPEED = 10
GAME_WIN_TILE = 2048
CUSTOM_GRID_SIZES = [4, 5, 6]
COLOR_THEMES = {
  'default': TILE_COLORS,
  'dark': {
    0: (60, 60, 60),
    2: (100, 100, 100),
```

```
    4: (120, 120, 120),
    8: (140, 140, 140),
    16: (160, 160, 160),
    32: (180, 180, 180),
    64: (200, 200, 200),
    128: (220, 220, 220),
    256: (240, 240, 240),
    512: (255, 255, 255),
    1024: (255, 255, 255),
    2048: (255, 255, 255),
  },
}
```

```
# Helper function to draw the grid

```

```
def draw_grid():
  for row in range(GRID_SIZE):
    for col in range(GRID_SIZE):
      pygame.draw.rect(screen, GRID_COLOR, [
        (GRID_MARGIN + TILE_SIZE) * col + GRID_MARGIN,
        (GRID_MARGIN + TILE_SIZE) * row + GRID_MARGIN,
        TILE_SIZE,
        TILE_SIZE
```

```
        ])

# Helper function to draw the tiles
```

```
def draw_tiles(grid):
  for row in range(GRID_SIZE):
    for col in range(GRID_SIZE):
      value = grid[row][col]
      if value != 0:
        pygame.draw.rect(screen, TILE_COLORS[value], [
          (GRID_MARGIN + TILE_SIZE) * col + GRID_MARGIN,
          (GRID_MARGIN + TILE_SIZE) * row + GRID_MARGIN,
          TILE_SIZE,
          TILE_SIZE
        ])
        text = font.render(str(value), True, FONT_COLOR)
        text_rect = text.get_rect(center=(
          (GRID_MARGIN + TILE_SIZE) * col +
          GRID_MARGIN + TILE_SIZE // 2,
          (GRID_MARGIN + TILE_SIZE) *
          row + GRID_MARGIN + TILE_SIZE // 2
        ))
        screen.blit(text, text_rect)
```

```
# Function to generate a new tile (2 or 4)
```

```
def generate_tile(grid):
    empty_cells = [(row, col) for row in range(GRID_SIZE)
                   for col in range(GRID_SIZE) if grid[row][col] == 0]
    if empty_cells:
        row, col = random.choice(empty_cells)
        grid[row][col] = random.choice([2, 4])

# Function to move the tiles in a given direction

```

```
def move_tiles(grid, direction):
    # Transpose the grid for easier handling of rows and columns
    if direction == 'left':
        grid = [list(row) for row in zip(*grid)]
    elif direction == 'up':
        grid = [list(col) for col in grid[::-1]]
    elif direction == 'down':
        grid = [list(col[::-1]) for col in grid]

    for row in range(GRID_SIZE):
        # Remove zeros
        non_zeros = [val for val in grid[row] if val != 0]
```

```
    # Merge tiles
    for col in range(len(non_zeros) - 1):
      if non_zeros[col] == non_zeros[col + 1]:
        non_zeros[col] *= 2
        non_zeros[col + 1] = 0
    # Remove zeros again
    non_zeros = [val for val in non_zeros if val != 0]
    # Fill the row with zeros
    grid[row] = non_zeros + [0] * (GRID_SIZE - len(non_zeros))

  # Undo the transposition
  if direction == 'left':
    grid = [list(row) for row in zip(*grid)]
  elif direction == 'up':
    grid = [list(col[::-1]) for col in grid[::-1]]
  elif direction == 'down':
    grid = [list(col) for col in grid[::-1]]

  return grid

# Function to check if the game is over
```

```
def is_game_over(grid):
  for row in grid:
```

```
    if 0 in row or any(row[i] == row[i + 1] for i in range(GRID_SIZE - 1)):
        return False
  for col in zip(*grid):
    if any(col[i] == col[i + 1] for i in range(GRID_SIZE - 1)):
        return False
  return True

# Function to display the game over screen
```

```
def game_over_screen():
  screen.fill(BACKGROUND_COLOR)
  game_over_text = font.render("Game Over!", True, FONT_COLOR)
  text_rect = game_over_text.get_rect(
    center=(SCREEN_SIZE[0] // 2, SCREEN_SIZE[1] // 2 - 30))
  screen.blit(game_over_text, text_rect)
  restart_text = font.render("Press R to Restart", True, FONT_COLOR)
  text_rect = restart_text.get_rect(
    center=(SCREEN_SIZE[0] // 2, SCREEN_SIZE[1] // 2 + 30))
  screen.blit(restart_text, text_rect)
  pygame.display.flip()

```

```
# Function to display the score
def display_score(score):
```

```
  score_text = font.render(f"Score: {score}", True, FONT_COLOR)
  screen.blit(score_text, (GRID_MARGIN, SCREEN_SIZE[1] - GRID_MARGIN - 30))

# Function to display the high score
```

```
def display_high_score(high_score):
  high_score_text = font.render(
    f"High Score: {high_score}", True, FONT_COLOR)
  screen.blit(high_score_text, (GRID_MARGIN,
          SCREEN_SIZE[1] - GRID_MARGIN - 60))
```

```
# Function to display the game win screen
def game_win_screen():
  screen.fill(BACKGROUND_COLOR)
  game_win_text = font.render("You Win!", True, FONT_COLOR)
  text_rect = game_win_text.get_rect(
    center=(SCREEN_SIZE[0] // 2, SCREEN_SIZE[1] // 2 - 30))
  screen.blit(game_win_text, text_rect)
  restart_text = font.render("Press R to Restart", True, FONT_COLOR)
  text_rect = restart_text.get_rect(
    center=(SCREEN_SIZE[0] // 2, SCREEN_SIZE[1] // 2 + 30))
  screen.blit(restart_text, text_rect)
  pygame.display.flip()
```

```
# Function to draw tiles with animations

def draw_animated_tiles(grid, animation_progress):
  for row in range(GRID_SIZE):
    for col in range(GRID_SIZE):
      value = grid[row][col]
      if value != 0:
        # Calculate the position with animation progress
        x = (GRID_MARGIN + TILE_SIZE) * col + GRID_MARGIN
        y = (GRID_MARGIN + TILE_SIZE) * row + GRID_MARGIN
        target_x = x + (TILE_SIZE + GRID_MARGIN) * animation_progress
        target_y = y + (TILE_SIZE + GRID_MARGIN) * animation_progress

        # Draw the tile with animation
        pygame.draw.rect(screen, TILE_COLORS[value], [
          target_x,
          target_y,
          TILE_SIZE,
          TILE_SIZE
        ])
        text = font.render(str(value), True, FONT_COLOR)
        text_rect = text.get_rect(center=(
          target_x + TILE_SIZE // 2,
```

```
            target_y + TILE_SIZE // 2
        ))
        screen.blit(text, text_rect)

# Function to move the tiles with animations
```

```
def move_animated_tiles(grid, direction, animation_progress):
    if direction == 'left':
        grid = [list(row) for row in zip(*grid)]
    elif direction == 'up':
        grid = [list(col) for col in grid[::-1]]
    elif direction == 'down':
        grid = [list(col[::-1]) for col in grid]

    for row in range(GRID_SIZE):
        non_zeros = [val for val in grid[row] if val != 0]
        for col in range(len(non_zeros) - 1):
            if non_zeros[col] == non_zeros[col + 1]:
                non_zeros[col] *= 2
                non_zeros[col + 1] = 0

        non_zeros = [val for val in non_zeros if val != 0]
        grid[row] = non_zeros + [0] * (GRID_SIZE - len(non_zeros))
```

```
    if direction == 'left':
        grid = [list(row) for row in zip(*grid)]
    elif direction == 'up':
        grid = [list(col[::-1]) for col in grid[::-1]]
    elif direction == 'down':
        grid = [list(col) for col in grid[::-1]]

    draw_animated_tiles(grid, animation_progress)
    pygame.display.flip()
    pygame.time.delay(ANIMATION_SPEED)

    return grid
```

```
# Function to animate tile movement

def animate_tile_movement(prev_grid, current_grid, direction, score, high_score):
    for i in range(1, ANIMATION_SPEED + 1):
        animation_progress = i / ANIMATION_SPEED
        screen.fill(BACKGROUND_COLOR)
        draw_grid()
        draw_animated_tiles(prev_grid, 1 - animation_progress)
        draw_animated_tiles(current_grid, animation_progress)
        display_score(score)
        display_high_score(high_score)
```

```
    pygame.display.flip()

  return current_grid
```

```
# Function to undo the last move
def undo_move(state_stack):
  if len(state_stack) > 1:
    state_stack.pop()  # Discard the current state
    prev_grid, prev_score, prev_high_score = state_stack.pop()
    screen.fill(BACKGROUND_COLOR)
    draw_grid()
    draw_tiles(prev_grid)
    display_score(prev_score)
    display_high_score(prev_high_score)
    pygame.display.flip()
    pygame.time.delay(ANIMATION_SPEED)

    return prev_grid, prev_score, prev_high_score
  else:
    return state_stack[0]  # If there's only one state, return it as is
```

```
# Main game loop
def main():
```

```
grid = [[0] * GRID_SIZE for _ in range(GRID_SIZE)]
generate_tile(grid)
generate_tile(grid)

running = True
game_over = False
game_won = False
score = 0
high_score = 0
state_stack = [([row[:] for row in grid], score, high_score)]

while running:
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      running = False
    elif event.type == pygame.KEYDOWN:
      if not game_over and not game_won:
        if event.key == pygame.K_LEFT:
          state_stack.append(
            ([row[:] for row in grid], score, high_score))
          grid = move_animated_tiles(grid, 'left', 0)
          grid = move_tiles(grid, 'left')
          if grid != state_stack[-1][0]:
            generate_tile(grid)
```

```
            score += calculate_score(state_stack[-1][0], grid)
            grid = animate_tile_movement(
              state_stack[-1][0], grid, 'left', score, high_score)
        elif event.key == pygame.K_RIGHT:
          state_stack.append(
            ([row[:] for row in grid], score, high_score))
          grid = move_animated_tiles(grid, 'right', 0)
          grid = move_tiles(grid, 'right')
          if grid != state_stack[-1][0]:
            generate_tile(grid)
            score += calculate_score(state_stack[-1][0], grid)
        elif event.key == pygame.K_UP:
          state_stack.append(
            ([row[:] for row in grid], score, high_score))
          grid = move_animated_tiles(grid, 'up', 0)
          grid = move_tiles(grid, 'up')
          if grid != state_stack[-1][0]:
            generate_tile(grid)
            score += calculate_score(state_stack[-1][0], grid)
        elif event.key == pygame.K_DOWN:
          state_stack.append(
            ([row[:] for row in grid], score, high_score))
          grid = move_animated_tiles(grid, 'down', 0)
          grid = move_tiles(grid, 'down')
```

```
            if grid != state_stack[-1][0]:
                generate_tile(grid)
                score += calculate_score(state_stack[-1][0], grid)
        elif event.key == pygame.K_r:
            grid = [[0] * GRID_SIZE for _ in range(GRID_SIZE)]
            generate_tile(grid)
            generate_tile(grid)
            game_over = False
            game_won = False
            if score > high_score:
                high_score = score
            score = 0
            state_stack = [
                ([row[:] for row in grid], score, high_score)]
        elif event.key == pygame.K_u and len(state_stack) > 1:
            grid, score, high_score = undo_move(state_stack)
            # Additional update to reflect the undone move
            game_over = False
            game_won = False

    elif game_over or game_won:
        if event.key == pygame.K_r:
            grid = [[0] * GRID_SIZE for _ in range(GRID_SIZE)]
            generate_tile(grid)
```

```
            generate_tile(grid)
            game_over = False
            game_won = False
            if score > high_score:
              high_score = score
            score = 0
            state_stack = [
              ([row[:] for row in grid], score, high_score)]

    if not game_over and not game_won:
      screen.fill(BACKGROUND_COLOR)
      draw_grid()
      draw_tiles(grid)
      display_score(score)
      display_high_score(high_score)
      pygame.display.flip()

      if is_game_over(grid):
        game_over = True
        if score > high_score:
          high_score = score
        game_over_screen()

      if GAME_WIN_TILE in [tile for row in grid for tile in row]:
```

```
            game_won = True
            game_win_screen()
            running = False

    pygame.quit()
```

```
def calculate_score(prev_grid, current_grid):
    score = 0
    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            if current_grid[i][j] > 0 and current_grid[i][j] != prev_grid[i][j]:
                score += current_grid[i][j]
    return score
```

```
if __name__ == "__main__":
    main()
```

This Python script implements a simple version of the popular game " 2048 " using the Pygame library . The game involves sliding numbered tiles on a 4x4 grid, merging tiles with the same number when they collide, and aiming to reach the tile with the number 2048 .

Let's break down the key components and functionalities of the script :

Pygame Initialization :

- The script starts by importing the Pygame library and initializing it .

Python Code

```
import pygame
```

Constants and Configuration :

- Various constants and configurations are set for the game, including grid size, tile size, colors, fonts, animation speed, winning condition, and color themes .

Python Code

```
GRID_SIZE = 4
TILE_SIZE = 100
GRID_MARGIN = 10
SCREEN_SIZE = (GRID_SIZE * TILE_SIZE + (GRID_SIZE + 1) * GRID_MARGIN,
               GRID_SIZE * TILE_SIZE + (GRID_SIZE + 1) * GRID_MARGIN)
BACKGROUND_COLOR = (187, 173, 160)
GRID_COLOR = (205, 193, 180)
FONT_COLOR = (255, 255, 255)
# ... other constants ...
```

Pygame Screen Initialization :

- The Pygame screen is initialized with the specified size and a caption .

Python Code

```
screen = pygame . display . set_mode ( SCREEN_SIZE )
pygame . display . set_caption (" 2048 Game ")
```

Grid and Tile Drawing :

- Functions for drawing the grid and tiles on the screen are **def**ined .

Python Code

```
def draw_grid ():
    # Draws the grid on the screen

def draw_tiles ( grid ):
    # Draws the tiles on the screen based on the provided grid
```

Tile Colors and Themes :

- Colors for each tile value and different color themes are **def**ined .

Python Code

```
TILE_COLORS = {
    # ... tile colors based on their values ...
```

```
}

COLOR_THEMES = {
    'default': TILE_COLORS,
    'dark': {
        # ... colors for a dark theme ...
    },
}
```

Tile Movement and Animation :

- Functions for moving tiles in different directions, generating new tiles, and handling animations are **def**ined .

Python Code

```
def move_tiles ( grid, direction ):
    # Moves the tiles in the specified direction

def generate_tile ( grid ):
    # Generates a new tile  ( 2 or 4 )  in an empty cell

def draw_animated_tiles ( grid, animation_progress ):
    # Draws tiles with animation based on the animation progress
```

Game State Management :

- Functions for checking game over conditions, displaying game over screens, and managing the game state are **def**ined .

Python Code

```
def is_game_over ( grid ):
    # Checks if the game is over

def game_over_screen ():
```

```
    # Displays the game over screen

def display_score ( score ):
    # Displays the current score on the screen
```

Main Game Loop :

- The main game loop handles user input, updates the game state, and continuously redraws the screen .

Python Code

```
def main ():
    # The main game loop
```

Additional Features :

- The script includes additional features such as undoing moves, calculating scores, and displaying a win screen .

Python Code

```
def undo_move ( state_stack ):
    # Undoes the last move

def calculate_score ( prev_grid, current_grid ):
    # Calculates the score based on changes in the grid
```

Running the Game :

- The script is executed when the file is run, starting the game .

Python Code

```
if __name__ == " __main__ ":
    main ()
```

Overall, this script provides a functional implementation of the 2048 game with basic features and Pygame for handling graphics and user input . Players can move tiles, merge them, achieve a winning condition, and restart the game .

How To Play 2048 Game

The 2048 game is a single - player sliding puzzle game where the goal is to combine matching tiles to reach the tile with a value of 2048 . The game is played on a 4x4 grid, and tiles with the same value can be merged by moving them in a specific direction . Here's a detailed breakdown of how to play the 2048 game based on the provided **Python Code** :

1. **Game Initialization :**

- The game starts with an empty 4x4 grid .
- Two tiles with a value of either 2 or 4 are randomly placed on the grid .

2. **Controls :**

- You can control the movement of tiles using the arrow keys on your keyboard .
- Press the **left arrow key** to move all tiles to the left .
- Press the **right arrow key** to move all tiles to the right .
- Press the **up arrow key** to move all tiles upwards .
- Press the **down arrow key** to move all tiles downwards .
- Press the **R key** to restart the game at any time .

3. **Gameplay :**

- Tiles with the same value can be merged into a single tile by moving them towards each other .
- When you make a move, a new tile with a value of 2 or 4 will appear in an empty spot .
- The goal is to keep merging tiles to create larger values and reach the tile with a value of 2048 .

- The game continues until you either reach the 2048 tile（winning the game）or the grid is full with no more moves possible（losing the game）.

4. **Scoring :**

- Your score is based on the values of the tiles you merge .
- Each time two tiles are merged, the value of the resulting tile is added to your score .

5. **Special Features :**

- The game includes a winning condition where reaching the 2048 tile displays a victory screen .
- There is an undo feature（press the **U key**）that allows you to undo your last move . However, it has limitations and can't be used in**def**initely .
- The game tracks and displays your current score and the highest score achieved in the session .

6. **Game Over :**

- The game ends when you reach the 2048 tile, displaying a victory screen .
- If the grid is full and no more moves are possible, the game ends, and a game over screen is displayed .
- You can restart the game at any time by pressing the **R key** .

7. **High Score :**

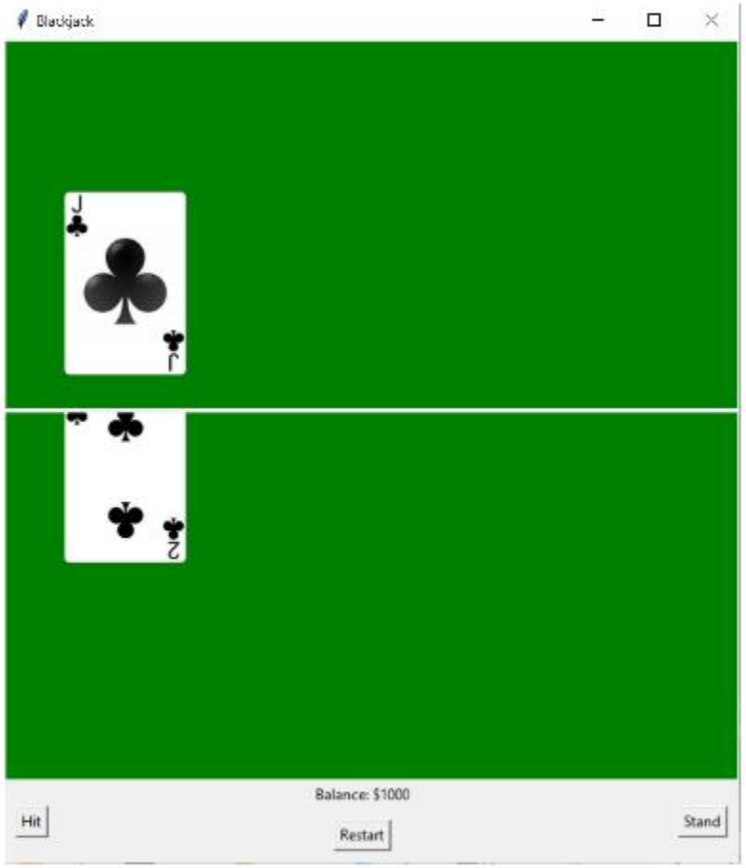
- The game keeps track of the highest score achieved during the session .

8. **Additional Details :**

- The grid and tiles are displayed using the Pygame library, and the game provides a graphical interface .

Overall, the 2048 game involves strategic thinking, planning your moves, and aiming to create the largest possible tiles to achieve the highest score . It's a simple yet challenging puzzle game that requires both skill and a bit of luck .

10. BlackJack Game



```
import tkinter as tk
from tkinter import messagebox
from PIL import Image, ImageTk
import random
```

```
class BlackjackGame:
  def __init__(self, master):
    self.master = master
    self.master.title("Blackjack")

    self.deck = self.get_deck()
    self.player_hand = []
    self.dealer_hand = []
    self.player_balance = 1000  # Initial balance

    self.player_card_images = []  # Track player card images
    self.dealer_card_images = []  # Track dealer card images

    self.create_widgets()

  def get_deck(self):
    suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
    ranks = ['2', '3', '4', '5', '6', '7',
         '8', '9', '10', 'J', 'Q', 'K', 'A']
```

```
    deck = [{'suit': suit, 'rank': rank}
            for suit in suits for rank in ranks]
    random.shuffle(deck)
    return deck

def deal_card(self, hand):
    card = self.deck.pop()
    hand.append(card)
    return card

def calculate_score(self, hand):
    score = sum(self.get_card_value(card) for card in hand)
    if score > 21 and 'A' in [card['rank'] for card in hand]:
        score -= 10  # Deduct 10 if there is an Ace and the score is over 21
    return score

def get_card_value(self, card):
    if card['rank'] in ['J', 'Q', 'K']:
        return 10
    elif card['rank'] == 'A':
        return 11
    else:
        return int(card['rank'])
```

```
def player_hit(self):
    # Disable the hit button during execution
    self.hit_button.config(state=tk.DISABLED)

    self.deal_card(self.player_hand)
    self.update_display()

    player_score = self.calculate_score(self.player_hand)
    if player_score > 21:
        self.end_game("You went over. You lose!")

    # Check for blackjack (10 and Ace)
    if player_score == 21 and len(self.player_hand) == 2:
        self.end_game("Blackjack! You win!")
    elif player_score <= 21:
        # Enable the hit button after execution if the score is still under 21
        self.hit_button.config(state=tk.NORMAL)
```

```
def end_game(self, message):
    if "You lose!" in message:
        self.player_balance -= 100  # Deduct 100 from balance when losing
        self.update_score()  # Update the score
```

```
    messagebox.showinfo("Game Over", message +
                        f"\nYour balance: ${self.player_balance}")
    self.update_score()  # Add parentheses to properly call the method

def dealer_play(self):
    while self.calculate_score(self.dealer_hand) < 17:
        self.deal_card(self.dealer_hand)
    self.update_display()

    player_score = self.calculate_score(self.player_hand)
    dealer_score = self.calculate_score(self.dealer_hand)

    if dealer_score > 21 or dealer_score < player_score:
        self.end_game("You win!")
        self.player_balance += 100
    elif dealer_score > player_score:
        self.end_game("You lose!")
        self.player_balance -= 100
    else:
        self.end_game("It's a draw!")

def restart_game(self):
    self.deck = self.get_deck()
    self.player_hand.clear()
```

```
        self.dealer_hand.clear()
        self.player_canvas.delete("all")
        self.dealer_canvas.delete("all")

        # Draw two initial cards for both player and dealer
        self.player_hand.extend([self.deal_card(self.player_hand), self.deal_card(self.player_hand)])
        self.dealer_hand.extend([self.deal_card(self.dealer_hand), self.deal_card(self.dealer_hand)])

        # Reset player balance to the initial value
        self.player_balance = 1000

        # Enable Hit and Stand buttons
        self.hit_button["state"] = "normal"
        self.stand_button["state"] = "normal"

        # Update the score label
        self.update_score()

        # Update the display
        self.update_display()
```

```
def _display_card(self, img_path, x, y, canvas, card_images=None):
    img_path = img_path.lower()
    image = Image.open(img_path)
```

```
    image = image.resize((100, 150), Image.ANTIALIAS)
    photo = ImageTk.PhotoImage(image)

    canvas.create_image(x, y, anchor=tk.W, image=photo)
    canvas.image = photo

    if card_images is not None:
        card_images.append(photo)
```

```
def update_display(self):
    self.update_player_hand()
    self.update_dealer_hand()
    self.update_score()  # Update score label after each action




def update_player_hand(self):
    self.player_canvas.delete("all")
    for card in self.player_hand:
        img_path = f"C:/Users/Suchat/Playing Cards/Playing Cards/PNG-cards-1.3/{card['rank']}_of_{card['suit']}.png"
        self._display_card(img_path, 50, 200, self.player_canvas, self.player_card_images)

def update_dealer_hand(self):
    self.dealer_canvas.delete("all")
```

```
    for card in self.dealer_hand:
        img_path = f"C:/Users/Suchat/Playing Cards/Playing Cards/PNG-cards-1.3/{card['rank']}_of_{card['suit']}.png"
        self._display_card(img_path, 50, 50, self.dealer_canvas, self.dealer_card_images)
```

```


def update_score(self):
    self.score_label.config(text=f"Balance: ${self.player_balance}")

def create_widgets(self):
    self.player_hand = [self.deal_card(
        self.player_hand), self.deal_card(self.player_hand)]
    self.dealer_hand = [self.deal_card(
        self.dealer_hand), self.deal_card(self.dealer_hand)]

    self.player_canvas = tk.Canvas(
        self.master, width=600, height=300, bg="green")
    self.player_canvas.pack()

    self.dealer_canvas = tk.Canvas(
        self.master, width=600, height=300, bg="green")
```

```
    self.dealer_canvas.pack()

    self.hit_button = tk.Button(
        self.master, text="Hit", command=self.player_hit)
    self.hit_button.pack(side=tk.LEFT, padx=10)

    self.stand_button = tk.Button(
        self.master, text="Stand", command=self.dealer_play)
    self.stand_button.pack(side=tk.RIGHT, padx=10)

    self.restart_button = tk.Button(
        self.master, text="Restart", command=self.restart_game)
    self.restart_button.pack(side=tk.BOTTOM, pady=10)

    self.score_label = tk.Label(
        self.master, text=f"Balance: ${self.player_balance}")
    self.score_label.pack(side=tk.BOTTOM)

    self.update_display()
```

```
def main():
    root = tk.Tk()
    game = BlackjackGame(root)
    root.mainloop()
```

```
if __name__ == "__main__":
    main()
```

This Python script uses the tkinter library to create a simple graphical user interface (GUI) for a Blackjack game . Let's go through the details of the script :

Import Statements :

Python Code

```
import tkinter as tk
from tkinter import messagebox
from PIL import Image, ImageTk
import random
```

- tkinter : The main library for creating the GUI .
- messagebox : A submodule of tkinter for displaying message boxes .
- Image and ImageTk from the PIL (Pillow) library for working with images .
- random : Used for shuffling the deck .

BlackjackGame Class :

Python Code

```
class BlackjackGame :
    def __init__ ( self, master ):
        # Initialization method
          # Initialize the main window and set the title
        self . master  =  master
        self . master . title (" Blackjack ")

        # Initialize the deck and player - related attributes
        self . deck  =  self . get_deck ()
        self . player_hand  = []
        self . dealer_hand  = []
        self . player_balance  =  1000

          # Track card images for player and dealer
        self . player_card_images  = []
        self . dealer_card_images  = []

        # Create GUI widgets
        self . create_widgets ()

    #  ... ( more methods explained below )
```

Methods :

1. get_deck : Creates and shuffles a standard deck of 52 playing cards .
2. deal_card : Deals a card from the deck and adds it to the specified hand .
3. calculate_score : Calculates the total score of a hand, considering the value of Aces .
4. get_card_value : Returns the numerical value of a card .
5. player_hit : Handles the logic when the player clicks the " Hit " button .
6. end_game : Displays a message box when the game ends, updating the player's balance .
7. dealer_play : Simulates the dealer's turn, continuing to draw cards until the score is 17 or higher .
8. restart_game : Resets the game state, allowing the player to start a new round .
9. _display_card : Displays a card image on a canvas at specified coordinates .
10. update_display : Updates the display after each player action .
11. update_player_hand : Updates the player's hand display on the GUI .
12. update_dealer_hand : Updates the dealer's hand display on the GUI .

13. update_score : Updates the displayed player balance .
14. create_widgets : Creates and configures the GUI elements .

GUI Elements :

- player_canvas and dealer_canvas : Canvas widgets for displaying player and dealer cards .
- hit_button and stand_button : Buttons for player actions .
- restart_button : Button to restart the game .
- score_label : Label to display the player's balance .

Main Function :

Python Code

```
def main ():
    root = tk . Tk ()
    game = BlackjackGame ( root )
    root . mainloop ()

if __name__ == " __main__ ":
    main ()
```

- Creates the main Tkinter window and initializes the BlackjackGame instance .
- Starts the Tkinter event loop .

Overall Flow :

1. The game starts with an initial deck, player hand, and dealer hand .
2. GUI elements are created and displayed .
3. Player actions (hit, stand) are handled through button clicks .
4. The game state and GUI are updated accordingly .
5. When the game ends, a message box is displayed with the result .

6. Player can restart the game for a new round .

The game combines basic Blackjack logic with a simple GUI for user interaction .

How To Play Blackjack Game

The provided code is a simple implementation of a Blackjack game using the Python Tkinter library for the graphical user interface (GUI). Let's break down the key components and explain how to play the game :

1. **Initialization :**
    - The game starts by creating an instance of the BlackjackGame class in the main function .
    - The game window is created using Tkinter .

Python Code

```
root = tk . Tk ()
game = BlackjackGame ( root )
root . mainloop ()
```

2. **Card Deck and Initialization :**
    - The get_deck method creates a standard deck of 52 playing cards, shuffles it, and returns the deck .
    - The player and dealer hands are initialized with two cards each .

Python Code

```
self . deck = self . get_deck ()
self . player_hand = [ self . deal_card ( self . player_hand ) ,
self . deal_card ( self . player_hand )]
self . dealer_hand = [ self . deal_card ( self . dealer_hand ) ,
self . deal_card ( self . dealer_hand )]
```

3. **Game Logic :**
    - The deal_card method deals a card from the deck to a specified hand .

- The calculate_score method calculates the total score of a given hand, considering the special case of Aces .
- The get_card_value method assigns numerical values to cards .
- The player_hit method handles player actions when they choose to hit, updating the display and checking for win / loss conditions .

Python Code

```
def player_hit ( self ):
    # ... ( disabled hit button during execution )
    self . deal_card ( self . player_hand )
    self . update_display ()
    player_score = self . calculate_score ( self . player_hand )
    if player_score > 21 :
        self . end_game (" You went over . You lose !")
    # ... ( checking for blackjack )
    elif player_score < = 21 :
        # ... ( enabled hit button after execution if the score is still under 21 )
```

4. **End Game Conditions :**

- The end_game method updates the player's balance based on win / loss conditions and displays a message box with the outcome .

Python Code

```
def end_game ( self, message ):
    # ... ( deduct 100 from balance when losing )
    messagebox . showinfo (" Game Over " , message + f " \nYour balance : ${self . player_balance} ")
    self . update_score ()
```

5. **Dealer's Turn :**

- The dealer_play method handles the dealer's turn, drawing cards until their score is at least 17 .
- It compares the final scores of the player and dealer to determine the game outcome .

Python Code

```
def dealer_play ( self ):
    # ... ( dealer draws cards until score is at least 17 )
    if dealer_score > 21 or dealer_score < player_score :
        self . end_game (" You win !")
        self . player_balance += 100
```

```
elif dealer_score > player_score :
    self . end_game (" You lose !")
    self . player_balance  -=  100
else :
    self . end_game (" It's a draw !")
```

6. **Restarting the Game :**
    - The `restart_game` method resets the game state, including the deck, hands, and player balance .

Python Code

```
def restart_game ( self ):
    # ... ( resetting game state )
    self . update_display ()
```

7. **GUI Elements :**
    - The GUI elements include two canvases for displaying player and dealer cards, " Hit " and " Stand " buttons for player actions, a " Restart " button, and a label showing the player's balance .

Python Code

```
self . player_canvas = tk . Canvas ( self . master, width = 600, height = 300, bg =" green ")
# ... ( similarly for dealer_canvas, hit_button, stand_button, restart_button, score_label )
```

8. **Card Display :**
    - The _display_card method handles the display of card images on the canvases .

Python Code

```
def _display_card ( self, img_path, x, y, canvas, card_images = None ):
    # ... ( opens and resizes the image, creates a Tkinter PhotoImage, and displays it on the
canvas )
```

9. **Updating the Display :**
    - The update_display method updates the player and dealer hands, as well as the score label after each action .

Python Code

```
def update_display ( self ):
    self . update_player_hand ()
    self . update_dealer_hand ()
    self . update_score ()
```

10. **Card Images :**
    - The card images are loaded from PNG files, and their paths are generated based on the card's rank and suit .

Python Code

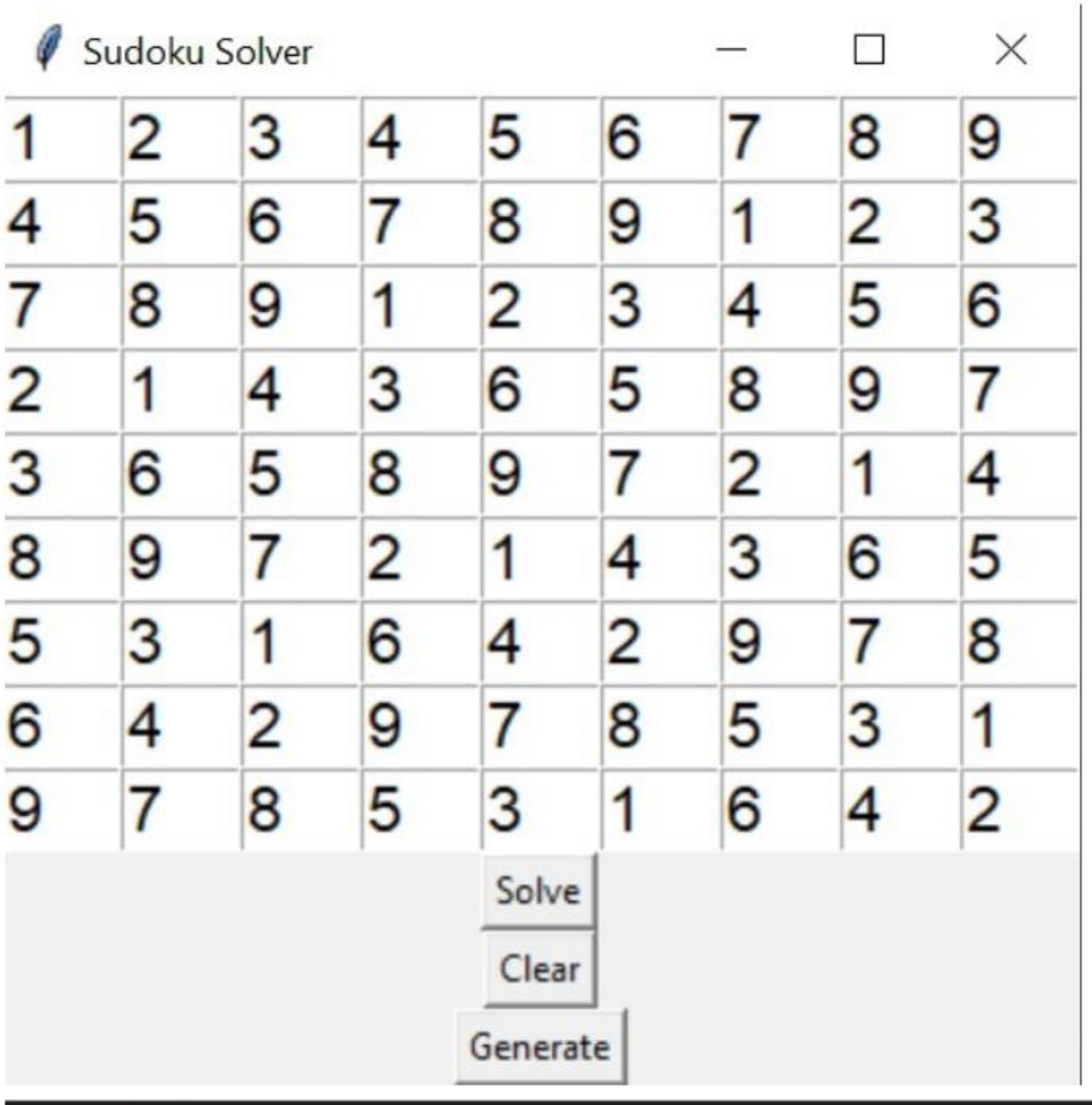
```
img_path = f " C :/ Users / Suchat / Playing Cards / Playing Cards / PNG - cards - 1 . 3 / {card [ 'rank' ] }_of_{card [ 'suit' ] } . png "
```

To play the game :

- Run the script .
- The initial player balance is $1000 .
- Click the " Hit " button to draw additional cards .
- Click the " Stand " button to end the player's turn and let the dealer play .
- The game automatically determines the winner and updates the balance .
- Click the " Restart " button to start a new game with a fresh deck .

Note : Ensure that the file paths for the card images are correct, and the image files are available in the specified directory .

11. Soduku Solver Game




```
import tkinter as tk
import random
```

```
class SudokuSolverGUI:
  def __init__(self, master):
    self.master = master
    self.master.title("Sudoku Solver")
    self.grid_size = 9
    self.cells = [[tk.StringVar() for _ in range(self.grid_size)]
             for _ in range(self.grid_size)]

    # Create the GUI grid
    for i in range(self.grid_size):
      for j in range(self.grid_size):
        cell_entry = tk.Entry(master, width=3, font=(
          'Arial', 16), textvariable=self.cells[i][j])
        cell_entry.grid(row=i, column=j)
        cell_entry.bind('<KeyRelease>', lambda event,
                i=i, j=j: self.update_entry_color(event, i, j))

    # Create Solve button
    solve_button = tk.Button(master, text="Solve",
                 command=self.solve_sudoku)
    solve_button.grid(row=self.grid_size, columnspan=self.grid_size)

    # Create Clear button
```

```
    clear_button = tk.Button(master, text="Clear", command=self.clear_grid)
    clear_button.grid(row=self.grid_size + 1, columnspan=self.grid_size)

    # Create Generate button
    generate_button = tk.Button(
        master, text="Generate", command=self.generate_board)
    generate_button.grid(row=self.grid_size + 2, columnspan=self.grid_size)

def on_key_press(self, event):
    # Limit entry to a single digit
    if event.char.isdigit() and int(event.char) in range(1, 10):
        event.widget.delete(0, tk.END)
        event.widget.insert(0, event.char)

    # Error highlighting
    row, col = self.get_cell_position(event.widget)
    num = int(event.char) if event.char.isdigit() else 0
    if not self.is_valid_move(row, col, num):
        event.widget.config(fg='red')
    else:
        event.widget.config(fg='black')

def clear_grid(self):
    # Clear all values in the grid
```

```
    for i in range(self.grid_size):
        for j in range(self.grid_size):
            self.cells[i][j].set("")
            # Instead of using config on StringVar, change the text color of Entry widget directly
            self.get_entry_widget(i, j).config(
                fg='black')  # Reset text color

def solve_sudoku(self):
    # Extract values from the GUI grid to create a Sudoku board
    board = [[0] * self.grid_size for _ in range(self.grid_size)]
    for i in range(self.grid_size):
        for j in range(self.grid_size):
            value = self.cells[i][j].get()
            if value.isdigit() and int(value) in range(1, 10):
                board[i][j] = int(value)

    if self.solve_sudoku_backtracking(board):
        # Update the GUI with the solved Sudoku
        for i in range(self.grid_size):
            for j in range(self.grid_size):
                self.cells[i][j].set(str(board[i][j]))

                # Instead of using config on StringVar, change the text color of Entry widget directly
                self.get_entry_widget(i, j).config(
```

```
                fg='black')  # Reset text color
        else:
            print("No solution found.")

    def update_entry_color(self, event, i, j):
        # Update the color of the Entry widget based on validity
        num = int(self.cells[i][j].get()
                  ) if self.cells[i][j].get().isdigit() else 0
        if not self.is_valid_move(self.cells, i, j, num):
            self.get_entry_widget(i, j).config(fg='red')
        else:
            self.get_entry_widget(i, j).config(fg='black')

    def get_entry_widget(self, i, j):
        # Get the Entry widget based on grid position
        widgets_in_row = self.master.grid_slaves(row=i)
        for widget in widgets_in_row:
            if int(widget.grid_info()["column"]) == j:
                return widget
        return None

    def is_valid_move(self, board, row, col, num):
        return not (
            self.used_in_row_backtracking(board, row, num) or
```

```
      self.used_in_col_backtracking(board, col, num) or
      self.used_in_box_backtracking(
        board, row - row % 3, col - col % 3, num)
    )

  def get_cell_position(self, widget):
    # Find the row and column of the given widget in the GUI grid
    for i in range(self.grid_size):
      for j in range(self.grid_size):
        if self.get_entry_widget(i, j) == widget:
          return i, j
    return -1, -1

  def find_unassigned_location(self, board):
    for i in range(self.grid_size):
      for j in range(self.grid_size):
        if board[i][j] == 0:
          return i, j
    return None

  def solve_sudoku_backtracking(self, board):
    empty_cell = self.find_unassigned_location(board)

    if not empty_cell:
```

```
        return True

    row, col = empty_cell

    for num in range(1, 10):
        if self.is_valid_move(board, row, col, num):
            board[row][col] = num

            if self.solve_sudoku_backtracking(board):
                return True

            board[row][col] = 0

    return False

  def used_in_row_backtracking(self, board, row, num):
    return num in board[row]

  def used_in_col_backtracking(self, board, col, num):
    return num in [board[row][col] for row in range(self.grid_size)]

  def used_in_box_backtracking(self, board, start_row, start_col, num):
    return any(num == board[row][col] for row in range(start_row, start_row + 3) for col in range(start_col, start_col +
3))
```

```
def generate_board(self):
    self.clear_grid()  # Clear the current grid

    # Generate a new Sudoku puzzle
    self.generate_sudoku_puzzle()

    # Display the generated puzzle on the GUI
    for i in range(self.grid_size):
        for j in range(self.grid_size):
            value = self.cells[i][j].get()
            if value.isdigit() and int(value) != 0:
                self.get_entry_widget(i, j).config(
                    fg='black')  # Reset text color

def generate_sudoku_puzzle(self):
    # Implement the logic to generate a new Sudoku puzzle
    # For simplicity, let's use a backtracking algorithm to fill in the board
    self.clear_grid()

    # Generate a solved Sudoku board
    solved_board = [[0] * self.grid_size for _ in range(self.grid_size)]
    self.solve_sudoku_backtracking(solved_board)
```

```
    # Create a copy of the solved board
    board_copy = [row[:] for row in solved_board]

    # Remove some numbers to create the puzzle
    # Adjust the number of cells to remove as needed
    cells_to_remove = random.randint(12, 30)
    for _ in range(cells_to_remove):
        row, col = random.randint(0, 8), random.randint(0, 8)
        while board_copy[row][col] == 0:
            row, col = random.randint(0, 8), random.randint(0, 8)
        board_copy[row][col] = ""

    # Update the GUI with the generated puzzle
    for i in range(self.grid_size):
        for j in range(self.grid_size):
            self.cells[i][j].set(str(board_copy[i][j]))
```

```
if __name__ == "__main__":
    root = tk.Tk()
    app = SudokuSolverGUI(root)
    root.mainloop()
```

let's go through the code line by line :

Python Code

```
import tkinter as tk
import random
```

- import tkinter as tk : This imports the Tkinter module and renames it as tk for easier reference .
- import random : This imports the random module, which will be used for generating random numbers later in the code .

Python Code

```
class SudokuSolverGUI :
    def __init__ ( self, master ):
        self . master = master
        self . master . title (" Sudoku Solver ")
        self . grid_size = 9
        self . cells = [[ tk . StringVar () for _ in range ( self . grid_size )]
                        for _ in range ( self . grid_size )]
```

- class SudokuSolverGUI : : This **def**ines a class named SudokuSolverGUI .
- **def** __init__ (self, master): : This is the constructor method for the class . It initializes the class instance with the given master widget (typically the root window of the GUI).
- self . master = master : This stores a reference to the master widget .

- self . master . title (" Sudoku Solver ") : This sets the title of the master widget to " Sudoku Solver ".
- self . grid_size = 9 : This sets the grid size for the Sudoku board to 9x9 .
- self . cells = [[tk . StringVar () for _ in range (self . grid_size)] for _ in range (self . grid_size)] : This creates a 2D list of StringVar objects, which will be used to represent the individual cells of the Sudoku grid .

Python Code

```
    # Create the GUI grid
for i in range ( self . grid_size ):
        for j in range ( self . grid_size ):
            cell_entry = tk . Entry ( master, width = 3, font =(
                'Arial', 16 ) , textvariable = self . cells [ i ][ j ])
            cell_entry . grid ( row = i, column = j )
            cell_entry . bind ( '<KeyRelease>', lambda event,
                                i = i, j = j : self . update_entry_color ( event, i, j ))
```

- This loop iterates over each cell in the Sudoku grid and creates an Entry widget for each cell .

- tk . Entry (master, width = 3, font =('Arial', 16) , textvariable = self . cells [i] [j]) : This creates an Entry widget with a width of 3 characters, using the Arial font with size 16, and associates it with a corresponding StringVar object from the self . cells list .
- cell_entry . grid (row = i, column = j) : This places the Entry widget in the grid layout at the specified row and column .
- cell_entry . bind ('<KeyRelease>', lambda event, i = i, j = j : self . update_entry_color (event, i, j)) : This binds the <KeyRelease> event to the update_entry_color method, passing the event object as well as the row (i) and column (j) indices as arguments .

Python Code

```
# Create Solve button
solve_button = tk . Button ( master, text =" Solve " ,
                             command = self . solve_sudoku )
solve_button . grid ( row = self . grid_size, columnspan = self . grid_size )
```

- This creates a " Solve " button widget using the tk . Button class, which calls the solve_sudoku method when clicked .

- solve_button . grid (row = self . grid_size, columnspan = self . grid_size) : This places the Solve button in the grid layout, spanning across the entire bottom row of the grid .

Python Code

```
# Create Clear button
clear_button = tk . Button ( master, text =" Clear " , command = self . clear_grid )
clear_button . grid ( row = self . grid_size + 1, columnspan = self . grid_size )
```

- This creates a " Clear " button widget using the tk . Button class, which calls the clear_grid method when clicked .
- clear_button . grid (row = self . grid_size + 1, columnspan = self . grid_size) : This places the Clear button in the grid layout, below the Solve button, spanning across the entire bottom row of the grid .

Python Code

```
# Create Generate button
generate_button = tk . Button (
    master, text =" Generate " , command = self . generate_board )
generate_button . grid ( row = self . grid_size + 2, columnspan = self . grid_size )
```

- This creates a " Generate " button widget using the tk . Button class, which calls the generate_board method when clicked .

- generate_button . grid (row = self . grid_size + 2, columnspan = self . grid_size) : This places the Generate button in the grid layout, below the Clear button, spanning across the entire bottom row of the grid .

This covers the initialization and setup of the GUI components in the SudokuSolverGUI class .

How To Play Sudoku Solver Game

To play the Sudoku Solver game, follow these steps :

1. **Run the Code :**
    - Make sure you have Python installed on your system .
    - Copy the provided **Python Code** into a file, save it with a . py extension, and run the script .
    - This will open a graphical window with the Sudoku Solver game .
2. **Understanding the GUI :**
    - The GUI consists of a 9x9 grid of Entry widgets, where you can input numbers .
    - The numbers represent the initial Sudoku puzzle, and you can edit them by clicking on the respective cells and typing in a digit from 1 to 9 .

3. **Solving the Puzzle :**

- If you have a Sudoku puzzle you'd like to solve, input the initial numbers into the grid .
- Click the " Solve " button to let the program solve the puzzle . The solved puzzle will be displayed in the same grid .

4. **Clearing the Grid :**

- To clear the entire grid, click the " Clear " button . This allows you to start with a blank grid or input a new puzzle .

5. **Generating a Puzzle :**

- Click the " Generate " button to create a new Sudoku puzzle . The program will generate a new puzzle for you to solve .

6. **Input Validation :**

- The program validates your input . If you enter an invalid number (e . g . , a number outside the range of 1 to 9) or if the entered number violates the rules of Sudoku, the text color will turn red to indicate an error .

7. **Error Highlighting :**

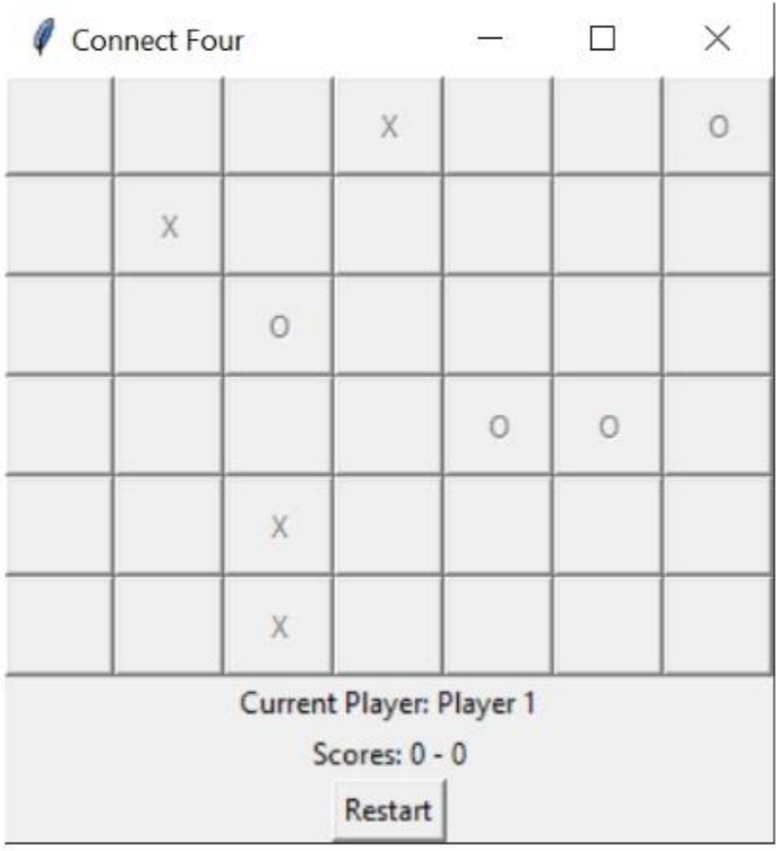
- As you input numbers, the program checks for errors and highlights the text in red if there's a conflict in the current row, column, or 3x3 box .

8. **Closing the Game :**

- Close the game window when you're done playing .

Remember, the Sudoku Solver game is designed to allow you to solve existing puzzles, clear the grid to create new ones, and enjoy the process of solving Sudoku puzzles interactively . You can experiment with different puzzles and observe how the backtracking algorithm efficiently solves them .

12. Connect Four Game



```
import tkinter as tk
from tkinter import messagebox
import random
```

```
class ConnectFour:
  def __init__(self, vs_ai=False):
    self.window = tk.Tk()
    self.window.title("Connect Four")
    self.player_names = ["Player 1", "Player 2"]
    self.scores = [0, 0]
    self.vs_ai = vs_ai
    self.board = [[0] * 7 for _ in range(6)]
    self.current_player = 1

    self.create_widgets()
    self.window.mainloop()

  def create_widgets(self):
    self.buttons = [[None] * 7 for _ in range(6)]

    for row in range(6):
      for col in range(7):
        self.buttons[row][col] = tk.Button(self.window, text="", width=5, height=2,
                                           command=lambda r=row, c=col: self.drop_piece(r, c))
```

```
        self.buttons[row][col].grid(row=row, column=col)

    self.player_label = tk.Label(
        self.window, text=f"Current Player: {self.player_names[self.current_player-1]}")
    self.player_label.grid(row=6, columnspan=7)

    self.score_label = tk.Label(
        self.window, text=f"Scores: {self.scores[0]} - {self.scores[1]}")
    self.score_label.grid(row=7, columnspan=7)

    restart_button = tk.Button(
        self.window, text="Restart", command=self.reset_game)
    restart_button.grid(row=8, columnspan=7)

def drop_piece(self, row, col):
    if self.board[row][col] == 0:
        self.board[row][col] = self.current_player
        self.update_button_text(row, col)
        if self.check_winner(row, col):
            messagebox.showinfo(
                "Winner!", f"Player {self.current_player} wins!")
            self.scores[self.current_player - 1] += 1
            self.update_score_label()
            self.reset_game()
```

```
        else:
            if not self.vs_ai:
                self.switch_player()
            else:
                self.ai_drop_piece()
                self.switch_player()

    def update_button_text(self, row, col):
        player_symbol = "X" if self.current_player == 1 else "O"
        self.buttons[row][col].config(text=player_symbol, state=tk.DISABLED)

    def check_winner(self, row, col):
        # right, down, diagonal right, diagonal left
        directions = [(0, 1), (1, 0), (1, 1), (-1, 1)]

        for dr, dc in directions:
            count = 1  # Number of consecutive pieces in the current direction

            for i in range(1, 4):
                r, c = row + i * dr, col + i * dc

                if 0 <= r < 6 and 0 <= c < 7 and self.board[r][c] == self.current_player:
                    count += 1
                else:
```

```
            break

        for i in range(1, 4):
            r, c = row - i * dr, col - i * dc

            if 0 <= r < 6 and 0 <= c < 7 and self.board[r][c] == self.current_player:
                count += 1
            else:
                break

        if count >= 4:
            return True

    return False

def switch_player(self):
    self.current_player = 3 - self.current_player  # Switch player between 1 and 2
    self.player_label.config(
        text=f"Current Player: {self.player_names[self.current_player-1]}")

def reset_game(self):
    for row in range(6):
        for col in range(7):
            self.buttons[row][col].config(text="", state=tk.NORMAL)
```

```
            self.board[row][col] = 0
        self.current_player = 1
        self.player_label.config(
            text=f"Current Player: {self.player_names[self.current_player-1]}")

    def update_score_label(self):
        self.score_label.config(
            text=f"Scores: {self.scores[0]} - {self.scores[1]}")

    def ai_drop_piece(self):
        valid_moves = [(r, c) for r in range(6)
                       for c in range(7) if self.board[r][c] == 0]
        if valid_moves:
            row, col = random.choice(valid_moves)
            self.board[row][col] = self.current_player
            self.update_button_text(row, col)
            if self.check_winner(row, col):
                messagebox.showinfo(
                    "Winner!", f"Player {self.current_player} wins!")
                self.scores[self.current_player - 1] += 1
                self.update_score_label()
                self.reset_game()
```

```
if __name__ == "__main__":
```

```
# Set vs_ai to False for two-player mode, or True for single-player vs AI mode
ConnectFour(vs_ai=True)
```

Let's go through the Connect Four game code line by line :

Python Code

```
import tkinter as tk
from tkinter import messagebox
import random
```

- These lines import the necessary modules for creating a graphical user interface（GUI） with Tkinter, handling message boxes, and generating random numbers .

Python Code

```
class ConnectFour :
    def __init__ ( self, vs_ai = False ):
```

- **Def**ines a class named ConnectFour that represents the Connect Four game. The vs_ai parameter is optional and set to False by **def**ault, indicating whether the game should be played against an AI.

Python Code

```
self . window = tk . Tk ()
self . window . title (" Connect Four ")
```

- Creates the main window for the game using Tkinter, with the title " Connect Four ".

Python Code

```
self . player_names = [" Player 1 " , " Player 2 "]
self . scores = [ 0, 0 ]
```

- Initializes a list of player names and a list to store the scores.

Python Code

```
self . vs_ai = vs_ai
```

- Stores the value of `vs_ai` in the class instance, determining whether the game is played against an AI or another player .

Python Code

```
self . board  = [[ 0 ] *  7 for _ in range ( 6 )]
```

- Initializes a 6x7 game board represented as a list of lists, where each element is initialized to 0 .

Python Code

```
self . current_player  =  1
```

- Sets the initial player to Player 1 .

Python Code

```
self . create_widgets ()
self . window . mainloop ()
```

- Calls the `create_widgets` method to set up the GUI components and starts the Tkinter event loop .

Python Code

```
def create_widgets ( self ):
```

- **Def**ines a method to create the GUI components of the game .

Python Code

```
self . buttons  = [[ None ] *  7 for _ in range ( 6 )]
```

- Initializes a 2D list `self . buttons` to store the button widgets in the GUI .

Python Code

```
for row in range ( 6 ):
        for col in range ( 7 ):
            self . buttons [ row ][ col ] =  tk . Button ( self . window, text ="" , width = 5,
height = 2,
                                                command = lambdar = row,      c = col :
self . drop_piece ( r, c ))
            self . buttons [ row ][ col ]. grid ( row = row, column = col )
```

- Creates a 6x7 grid of buttons, where each button is associated with the `drop_piece` method using the `command` parameter . The buttons are displayed in the Tkinter window using the `grid` method .

Python Code

```
self . player_label  =  tk . Label (
        self . window,                          text = f " Current                          Player :
{self . player_names [ self . current_player - 1 ] } ")
self . player_label . grid ( row = 6, columnspan = 7 )
```

- Creates a label to display the current player's turn and places it at the bottom of the window .

Python Code

```
self . score_label  =  tk . Label (
        self . window, text = f " Scores :  {self . scores [ 0 ] } -  {self . scores [ 1 ] } ")
self . score_label . grid ( row = 7, columnspan = 7 )
```

- Creates a label to display the scores and places it below the player label .

Python Code

```
restart_button  =  tk . Button (
        self . window, text =" Restart " , command = self . reset_game )
restart_button . grid ( row = 8, columnspan = 7 )
```

- Creates a restart button and associates it with the reset_game method .

Python Code

```
def drop_piece ( self, row, col ):
```

- **Def**ines the drop_piece method, which is called when a player clicks a button to drop a game piece .

Python Code

```
if self . board [ row ][ col ] ==  0 :
        self . board [ row ][ col ] =  self . current_player
        self . update_button_text ( row, col )
```

- Checks if the selected cell is empty, then updates the game board and button text with the current player's move .

Python Code

```
if self . check_winner ( row, col ):
    messagebox . showinfo (
        " Winner !" , f " Player {self . current_player} wins !")
    self . scores [ self . current_player  -  1 ] +=  1
    self . update_score_label ()
    self . reset_game ()
```

- Checks if the current move results in a win . If so, displays a message box, updates the scores, and resets the game .

Python Code

```
else :
    if not self . vs_ai :
        self . switch_player ()
    else :
        self . ai_drop_piece ()
        self . switch_player ()
```

- If the game is not against an AI, switches to the next player . Otherwise, the AI makes a move, and then the player is switched .

Python Code

```
def update_button_text ( self, row, col ):
```

- **Def**ines a method to update the button text with the player's symbol .

Python Code

```
player_symbol = " X " if self . current_player == 1 else " O "
self . buttons [ row ][ col ]. config ( text = player_symbol, state = tk . DISABLED )
```

- Determines the player's symbol based on the current player and updates the button text . Disables the button to prevent further moves in that cell .

Python Code

```
def check_winner ( self, row, col ):
```

- **Def**ines the check_winner method to determine if a player has won the game .

Python Code

```
directions = [( 0, 1 ), ( 1, 0 ), ( 1, 1 ), (- 1, 1 )]
```

- **Def**ines four possible directions to check for winning combinations： right, down, diagonal right, and diagonal left .

Python Code

```
for dr, dc in directions :
        count = 1 # Number of consecutive pieces in the current direction
```

- Iterates through each direction and initializes a counter for consecutive pieces .

Python Code

```
        for i in range ( 1, 4 ):
            r, c = row + i * dr, col + i * dc

            if 0< = r < 6 and 0< = c < 7 and self . board [ r ][ c ] == self . current_player :
                count += 1
            else :
                break
```

- Checks for consecutive pieces in the positive direction of the current direction .

Python Code

```
        for i in range ( 1, 4 ):
            r, c = row - i * dr, col - i * dc

            if 0< = r < 6 and 0< = c < 7 and self . board [ r ][ c ] == self . current_player :
                count += 1
```

```
else :
    break
```

- Checks for consecutive pieces in the negative direction of the current direction .

Python Code

```
if count > = 4 :
    return True
```

- If there are four or more consecutive pieces in any direction, the player wins .

Python Code

```
return False
```

- If no winning combination is found, returns False .

Python Code

```
def switch_player ( self ):
```

- **Def**ines the switch_player method to switch between players .

Python Code

```
self . current_player = 3 - self . current_player # Switch player between 1 and 2
self . player_label . config (
    text = f " Current Player : {self . player_names [ self . current_player - 1 ] } ")
```

- Toggles between players 1 and 2 and updates the player label accordingly .

Python Code

```
def reset_game ( self ):
```

- **Def**ines the reset_game method to reset the game board and player turns .

Python Code

```
for row in range ( 6 ):
        for col in range ( 7 ):
            self . buttons [ row ][ col ]. config ( text ="" , state = tk . NORMAL )
            self . board [ row ][ col ] =  0
self . current_player  =  1
self . player_label . config (
        text = f " Current Player :  {self . player_names [ self . current_player - 1 ] } ")
```

- Clears the button texts, enables the buttons, resets the game board, sets the current player to 1, and updates the player label .

Python Code

```
def update_score_label ( self ):
```

- **Def**ines the update_score_label method to update the displayed scores .

Python Code

```
self . score_label . config (
        text = f " Scores :  {self . scores [ 0 ] } -  {self . scores [ 1 ] } ")
```

- Updates the score label with the current scores .

Python Code

```
def ai_drop_piece ( self ):
```

- **Def**ines the ai_drop_piece method for the AI to make a move .

Python Code

```
valid_moves  = [( r, c )  for r in range ( 6 )
                    for c in range ( 7 )  if self . board [ r ][ c ] ==  0 ]
```

- Creates a list of valid moves (empty cells) for the AI to choose from .

Python Code

```
if valid_moves :
        row, col  =  random . choice ( valid_moves )
        self . board [ row ][ col ] =  self . current_player
        self . update_button_text ( row, col )
```

- If there are valid moves, the AI randomly selects a move, updates the game board, and updates the button text .

Python Code

```
            if self . check_winner ( row, col ):
                messagebox . showinfo (
                    " Winner !" , f " Player {self . current_player} wins !")
                self . scores [ self . current_player  -  1 ] +=  1
                self . update_score_label ()
                self . reset_game ()
```

- Checks if the AI move results in a win and handles the game outcome accordingly .

Python Code

```
if __name__  == " __main__ ":
    ConnectFour ( vs_ai = True )
```

- If the script is run as the main program, it creates an instance of the ConnectFour class with AI mode enabled .

How To Play Connect Four Game

Connect Four is a two - player strategy game where the objective is to connect four of your own game pieces in a row, either horizontally, vertically, or diagonally, before your opponent does . Here's a step - by - step guide on how to play Connect Four :

1 . Setup :

- The game is typically played on a 6x7 grid, though the size can vary .
- Each player is assigned a color (commonly red and yellow) or a symbol (like " X " and " O ").
- The game starts with an empty grid .

2 . Starting Player :

- Players decide who goes first . This can be done through a coin toss, rock - paper - scissors, or any other agreed - upon method .

3 . Taking Turns :

- Players take turns dropping one of their pieces into any of the seven columns on the grid .
- The piece will fall to the lowest available space in the chosen column .

4 . Goal :

- The goal is to be the first to connect four of your pieces in a row, either horizontally, vertically, or diagonally .

5 . Winning :

- The game ends as soon as one player successfully connects four of their pieces in a row .
- The winning player should announce their victory, and the game concludes .

6 . Draw :

- If the entire grid is filled without any player achieving four in a row, the game is a draw .

7 . Restarting :

- After a game concludes, players may decide to play again by resetting the board .

8 . Strategies :

- Pay attention to your opponent's moves to block potential winning combinations .
- Plan ahead and consider creating multiple opportunities to win simultaneously .
- Be cautious not to create opportunities for your opponent to win .

9 . Variant : AI Mode (if applicable):

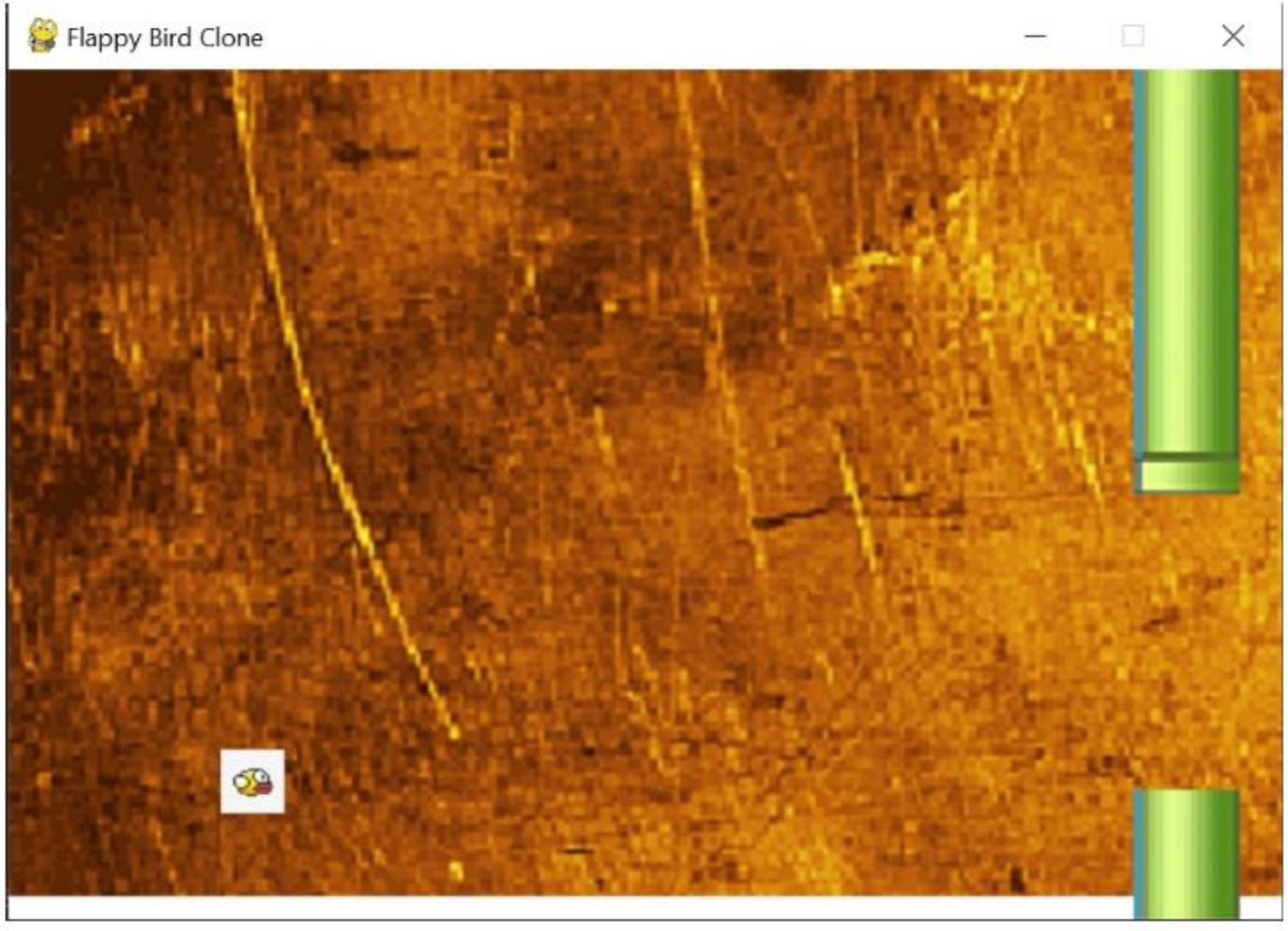
- If playing against an AI, the computer will make its moves based on its programmed strategy .
- Follow the same rules, but be prepared for strategic decisions from the AI opponent .

10 . Enjoyment :

- Connect Four is a fun and quick game . Focus on enjoying the strategic aspects and friendly competition .

Remember that Connect Four is a game of skill and strategy, and with practice, you can improve your ability to anticipate and block your opponent's moves while creating winning opportunities for yourself .

13. Flappy Bird Clone Game



```
import pygame
import sys
import random

# Initialize Pygame
pygame.init()

# Constants
WIDTH, HEIGHT = 600, 400
FPS = 30  # Adjust the frame rate
GRAVITY = 1.0  # Adjust the gravity
JUMP_HEIGHT = 10  # Adjust the jump height
PIPE_WIDTH = 50
PIPE_HEIGHT = 200
PIPE_GAP = 250  # Adjust the gap between pipes

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)

# Create the game window
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Flappy Bird Clone")
```

```
# Load images
bird_image = pygame.image.load("bird.png")
background_image = pygame.image.load("background.png")
pipe_image = pygame.image.load("pipe.png")

# Scale images
bird_image = pygame.transform.scale(
  bird_image, (30, 30)) # Adjust the bird size
pipe_image = pygame.transform.scale(pipe_image, (PIPE_WIDTH, PIPE_HEIGHT))

# Clock to control the frame rate
clock = pygame.time.Clock()
```

```
class Bird:
  def __init__(self):
    self.x = 100
    self.y = HEIGHT // 2
    self.width = 30  # Adjust the bird's width
    self.height = 30  # Adjust the bird's height
    self.velocity = 0

  def jump(self):
    self.velocity = -JUMP_HEIGHT
```

```
  def update(self):
    self.velocity += GRAVITY
    self.y += self.velocity

    # Keep the bird within the screen bounds
    if self.y < 0:
      self.y = 0
    if self.y > HEIGHT - self.height:
      self.y = HEIGHT - self.height

  def draw(self):
    screen.blit(bird_image, (self.x, self.y))
```

```
class Pipe:
  def __init__(self, x):
    self.x = x
    self.gap_top = random.randint(50, HEIGHT - PIPE_GAP - 50)

  def update(self):
    self.x -= 5

  def draw(self):
    screen.blit(pipe_image, (self.x, 0))
    lower_pipe_top = self.gap_top + PIPE_GAP
```

```
    screen.blit(pipe_image, (self.x, lower_pipe_top))
```

```
# Create objects
bird = Bird()
pipes = []

# Main game loop
while True:
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      pygame.quit()
      sys.exit()
    elif event.type == pygame.KEYDOWN:
      if event.key == pygame.K_SPACE:
        bird.jump()

  # Update objects
  bird.update()
  for pipe in pipes:
    pipe.update()

  # Create new pipes
  if len(pipes) == 0 or pipes[-1].x < WIDTH - 200:
    pipes.append(Pipe(WIDTH))
```

```
# Remove off-screen pipes
pipes = [pipe for pipe in pipes if pipe.x > -PIPE_WIDTH]

# Collision detection
for pipe in pipes:
    bird_rect = pygame.Rect(bird.x, bird.y, bird.width, bird.height)
    upper_pipe_rect = pygame.Rect(pipe.x, 100, PIPE_WIDTH, pipe.gap_top)
    lower_pipe_top = pipe.gap_top + PIPE_GAP
    lower_pipe_rect = pygame.Rect(
        pipe.x, lower_pipe_top+5, PIPE_WIDTH, HEIGHT - lower_pipe_top)

    # Check for collision with upper and lower pipes
    if bird_rect.colliderect(upper_pipe_rect) or bird_rect.colliderect(lower_pipe_rect):
        print("Game Over")
        bird = Bird()  # Reset the bird position
        pipes = []    # Reset the pipes
        pygame.time.delay(1000)  # Pause for a moment before restarting

# Draw background
screen.fill(WHITE)
screen.blit(background_image, (0, 0))

# Draw objects
```

```
bird.draw()
for pipe in pipes:
  pipe.draw()

# Update display
pygame.display.flip()

# Control the frame rate
clock.tick(FPS)
```

let's go through the provided code line by line to understand its functionality :

Python Code

```
import pygame
import sys
import random
```

These lines import the necessary libraries for the game : pygame for game development, sys for system - related functionality, and random for generating random numbers .

Python Code

```
# Initialize Pygame
pygame . init ()
```

This line initializes the Pygame library .

Python Code

```
# Constants
WIDTH, HEIGHT = 600, 400
FPS = 30 # Adjust the frame rate
GRAVITY = 1.0 # Adjust the gravity
JUMP_HEIGHT = 10 # Adjust the jump height
PIPE_WIDTH = 50
PIPE_HEIGHT = 200
PIPE_GAP = 250 # Adjust the gap between pipes
```

Here, various constants are **def**ined, such as the width and height of the game window, frame rate (FPS) , gravity for the bird's movement, jump height, and dimensions for the pipes .

Python Code

```
# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
```

These lines **def**ine color constants using RGB values .

Python Code

```
# Create the game window
screen = pygame . display . set_mode (( WIDTH, HEIGHT ))
pygame . display . set_caption (" Flappy Bird Clone ")
```

These lines create the game window with the specified width and height and set the window caption .

Python Code

```
# Load images
bird_image = pygame . image . load (" bird . png ")
background_image = pygame . image . load (" background . png ")
pipe_image = pygame . image . load (" pipe . png ")
```

These lines load the images for the bird, background, and pipe from files .

Python Code

```
# Scale images
bird_image = pygame . transform . scale ( bird_image, ( 30, 30 )) # Adjust the bird size
pipe_image = pygame . transform . scale ( pipe_image, ( PIPE_WIDTH, PIPE_HEIGHT ))
```

The images are scaled to the desired dimensions .

Python Code

```
# Clock to control the frame rate
clock = pygame . time . Clock ()
```

A clock object is created to control the frame rate .

Python Code

```
class Bird :
    def __init__ ( self ):
        self . x = 100
        self . y = HEIGHT // 2
        self . width = 30 # Adjust the bird's width
        self . height = 30 # Adjust the bird's height
        self . velocity = 0

    def jump ( self ):
        self . velocity = - JUMP_HEIGHT

    def update ( self ):
        self . velocity += GRAVITY
        self . y += self . velocity

          # Keep the bird within the screen bounds
```

```
        if self . y < 0 :
                self . y = 0
        if self . y > HEIGHT - self . height :
                self . y = HEIGHT - self . height

    def draw ( self ):
        screen . blit ( bird_image, ( self . x, self . y ))
```

A class Bird is **def**ined to represent the player - controlled bird . It has methods for jumping, updating its position based on gravity, and drawing the bird on the screen .

Python Code

```
class Pipe :
    def __init__ ( self, x ):
        self . x = x
        self . gap_top = random . randint ( 50, HEIGHT - PIPE_GAP - 50 )

    def update ( self ):
        self . x -= 5

    def draw ( self ):
        screen . blit ( pipe_image, ( self . x, 0 ))
```

```
        lower_pipe_top = self.gap_top + PIPE_GAP
        screen.blit(pipe_image, (self.x, lower_pipe_top))
```

A class Pipe is **def**ined to represent the pipes in the game. Pipes are initialized with a random gap position, and they move from right to left (update method) as the game progresses. The draw method is responsible for rendering the upper and lower pipes on the screen.

Python Code

```
# Create objects
bird = Bird()
pipes = []
```

Instances of the Bird and an empty list for pipes are created.

Python Code

```
# Main game loop
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
```

```
    elif event . type ==  pygame . KEYDOWN :
            if event . key  ==  pygame . K_SPACE :
                bird . jump ()
```

The main game loop begins, which continuously checks for events such as quitting the game or pressing the space key to make the bird jump .

Python Code

```
    # Update objects
  bird . update ()
  for pipe in pipes :
    pipe . update ()
```

The positions of the bird and pipes are updated in each iteration of the game loop .

Python Code

```
    # Create new pipes
  if len ( pipes ) ==  0 or pipes [- 1 ]. x < WIDTH  -  200 :
    pipes . append ( Pipe ( WIDTH ))
```

New pipes are created if there are no pipes or if the last pipe's x - coordinate is beyond a certain threshold .

Python Code

```
    # Remove off - screen pipes
    pipes  = [ pipe for pipe in pipes if pipe . x >  - PIPE_WIDTH ]
```

Pipes that have moved off - screen are removed from the list .

Python Code

```
     # Collision detection
    for pipe in pipes :
        bird_rect  =  pygame . Rect ( bird . x, bird . y, bird . width, bird . height )
        upper_pipe_rect  =  pygame . Rect ( pipe . x, 100, PIPE_WIDTH, pipe . gap_top )
        lower_pipe_top  =  pipe . gap_top  +  PIPE_GAP
        lower_pipe_rect  =  pygame . Rect (
                pipe . x, lower_pipe_top + 5, PIPE_WIDTH, HEIGHT  -  lower_pipe_top )

        # Check for collision with upper and lower pipes
        if                          bird_rect . colliderect ( upper_pipe_rect )                  or
bird_rect . colliderect ( lower_pipe_rect ):
                print (" Game Over ")
                bird  =  Bird ()  # Reset the bird position
                pipes  = []     # Reset the pipes
                pygame . time . delay ( 1000 )  # Pause for a moment before restarting
```

Collision detection is performed to check if the bird collides with the upper or lower pipes . If a collision occurs, the game prints " Game Over, " resets the bird's position, clears the pipes, and introduces a delay before restarting .

Python Code

```
    # Draw background
    screen . fill ( WHITE )
    screen . blit ( background_image, ( 0, 0 ))
```

The background is filled with the WHITE color, and the background image is drawn on the screen .

Python Code

```
    # Draw objects
    bird . draw ()
    for pipe in pipes :
        pipe . draw ()
```

The bird and pipes are drawn on the screen .

Python Code

```
    # Update display
    pygame . display . flip ()
```

The display is updated .

Python Code

```
    # Control the frame rate
   clock . tick ( FPS )
```

The frame rate is controlled to match the specified FPS .

This code creates a simple Flappy Bird clone using the Pygame library, with a bird that can jump and avoid pipes . The game loop continuously updates the game state, checks for collisions, and handles user input .

How To Play Flappy Bird Clone

To play the Flappy Bird clone you've provided, follow these steps :

1. **Run the Code :**
    - Make sure you have Python and Pygame installed on your system .
    - Save the provided code in a Python file, for example, " flappy_bird_clone . py ".
    - Open a terminal or command prompt, navigate to the directory containing the Python file, and run it using python flappy_bird_clone . py .
2. **Game Controls :**

- Press the **Spacebar** to make the bird jump .

3. **Game Objective :**

- Navigate the bird through the gaps between the pipes without hitting them .

4. **Game Mechanics :**

- The bird will fall due to gravity, and you must use the spacebar to make it jump .
- The pipes move from right to left, and new pipes are generated periodically .
- The goal is to keep the bird flying and pass through as many gaps between pipes as possible .

5. **Game Over :**

- The game ends if the bird collides with the pipes .
- When a collision occurs, the game will print " Game Over, " reset the bird's position, clear existing pipes, and pause for a moment before restarting .

6. **Repeat :**

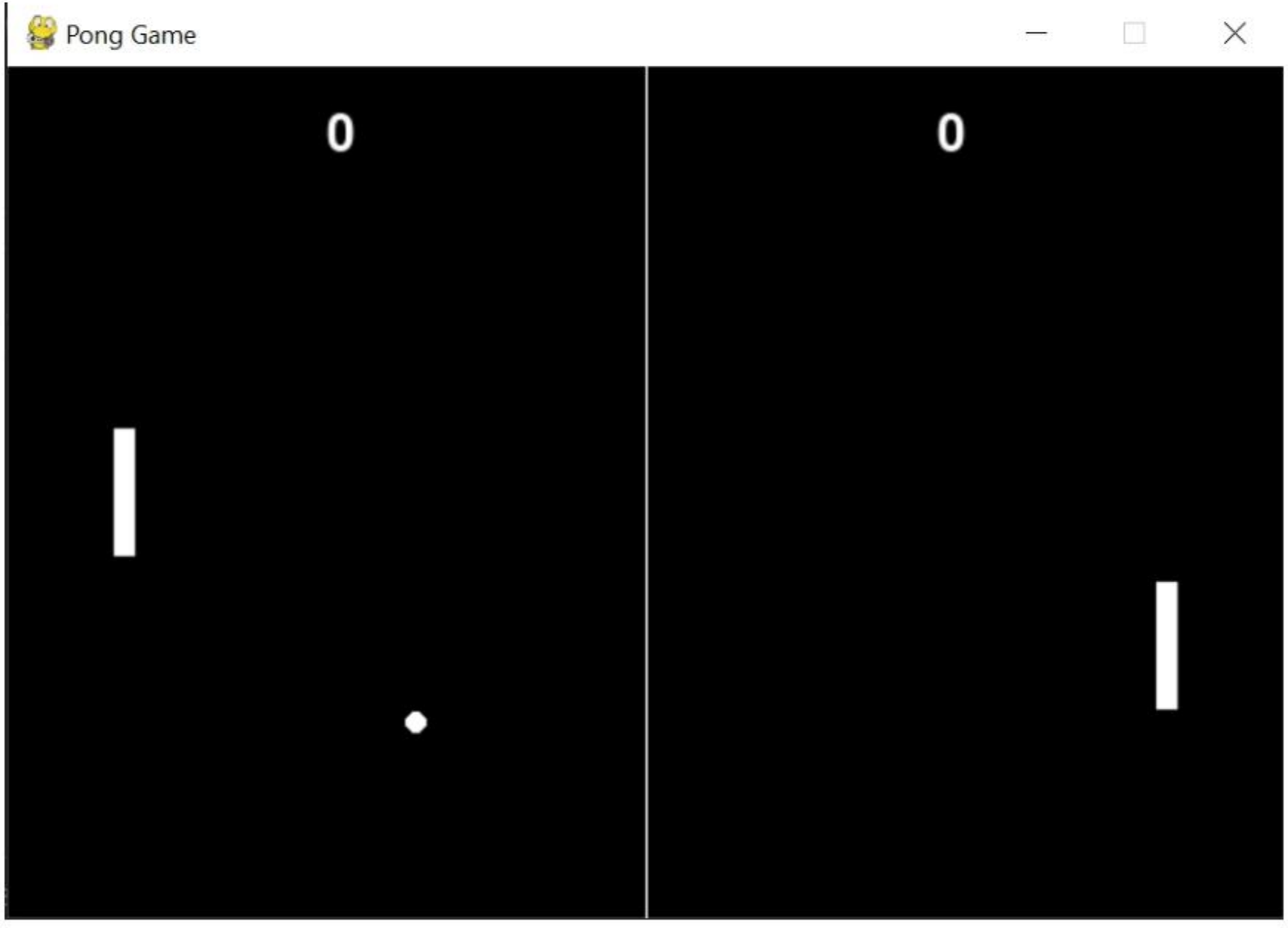
- The game will continue to run in a loop, allowing you to play again after each Game Over .

7. **Adjustments :**

- You can modify constants in the code (such as GRAVITY , JUMP_HEIGHT , FPS , etc .) to change the difficulty or behavior of the game .

Remember that Flappy Bird is known for its challenging and addictive gameplay . Try to beat your high score by maneuvering the bird through the pipes with well - timed jumps . Good luck and have fun playing your Flappy Bird clone !

14. Pong Game



```
import pygame
import sys
import random

# Initialize Pygame
pygame.init()

# Constants
WIDTH, HEIGHT = 600, 400
BALL_RADIUS = 10
PADDLE_WIDTH, PADDLE_HEIGHT = 10, 60
FPS = 60
WHITE = (255, 255, 255)

# Create screen
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Pong Game")

# Create paddles and ball
player_paddle = pygame.Rect(
    50, HEIGHT // 2 - PADDLE_HEIGHT // 2, PADDLE_WIDTH, PADDLE_HEIGHT)
opponent_paddle = pygame.Rect(
    WIDTH - 50 - PADDLE_WIDTH, HEIGHT // 2 - PADDLE_HEIGHT // 2, PADDLE_WIDTH, PADDLE_HEIGHT)
ball = pygame.Rect(WIDTH // 2 - BALL_RADIUS // 2, HEIGHT //
```

```
            2 - BALL_RADIUS // 2, BALL_RADIUS, BALL_RADIUS)

# Initialize velocities
ball_speed_x = 3 * random.choice([1, -1])
ball_speed_y = 3 * random.choice([1, -1])
player_speed = 0
opponent_speed = 2  # Adjust opponent's paddle speed

# Initialize score
player_score = 0
opponent_score = 0

# Function to reset the ball's position
```

```
def reset_ball():
    ball.x = WIDTH // 2 - BALL_RADIUS // 2
    ball.y = HEIGHT // 2 - BALL_RADIUS // 2
```

```
# Game loop
clock = pygame.time.Clock()

while True:
    for event in pygame.event.get():
```

```
    if event.type == pygame.QUIT:
        pygame.quit()
        sys.exit()
    elif event.type == pygame.KEYDOWN:
        if event.key == pygame.K_UP:
            player_speed = -3
        elif event.key == pygame.K_DOWN:
            player_speed = 3
        elif event.key == pygame.K_r:
            # Restart the game when 'R' is pressed
            player_score = 0
            opponent_score = 0
            player_paddle.y = HEIGHT // 2 - PADDLE_HEIGHT // 2
            reset_ball()
            ball_speed_x = 3 * random.choice([1, -1])
            ball_speed_y = 3 * random.choice([1, -1])
    elif event.type == pygame.KEYUP:
        if event.key == pygame.K_UP or event.key == pygame.K_DOWN:
            player_speed = 0

# Move paddles and ball
player_paddle.y += player_speed
# Ensure the player's paddle stays within the game window
player_paddle.y = max(0, min(player_paddle.y, HEIGHT - PADDLE_HEIGHT))
```

```
ball.x += ball_speed_x
ball.y += ball_speed_y

# Ball collisions with walls
if ball.top <= 0 or ball.bottom >= HEIGHT:
    ball_speed_y = -ball_speed_y

# Ball collisions with paddles
if ball.colliderect(player_paddle):
    ball_speed_x = abs(ball_speed_x)  # Change direction
elif ball.colliderect(opponent_paddle):
    ball_speed_x = -abs(ball_speed_x)  # Change direction

# Check if the ball passed the paddles
if ball.left <= 0:
    opponent_score += 1  # Increase opponent score
    reset_ball()
elif ball.right >= WIDTH:
    player_score += 1  # Increase player score
    reset_ball()

# Opponent AI
if opponent_paddle.centery < ball.centery:
```

```
    opponent_paddle.y += min(opponent_speed,
                             ball.centery - opponent_paddle.centery)
elif opponent_paddle.centery > ball.centery:
    opponent_paddle.y -= min(opponent_speed,
                             opponent_paddle.centery - ball.centery)

# Draw everything
screen.fill((0, 0, 0))
pygame.draw.rect(screen, WHITE, player_paddle)
pygame.draw.rect(screen, WHITE, opponent_paddle)
pygame.draw.ellipse(screen, WHITE, ball)

# Draw the middle line
pygame.draw.aaline(screen, WHITE, (WIDTH // 2, 0), (WIDTH // 2, HEIGHT))

# Draw scores
font = pygame.font.Font(None, 36)
player_text = font.render(str(player_score), True, WHITE)
opponent_text = font.render(str(opponent_score), True, WHITE)
screen.blit(player_text, (WIDTH // 4, 20))
screen.blit(opponent_text, (3 * WIDTH // 4 -
        opponent_text.get_width(), 20))

# Update the display
```

```
pygame.display.flip()

# Cap the frame rate
clock.tick(FPS)
```

let's go through the code line by line to understand each part :

Python Code

```
import pygame
import sys
import random
```

Here, the code imports the necessary modules : pygame for creating the game, sys for handling system - related operations, and random for generating random numbers .

Python Code

```
pygame . init ()
```

This line initializes the Pygame library .

Python Code

```
# Constants
WIDTH, HEIGHT = 600, 400
BALL_RADIUS = 10
```

```
PADDLE_WIDTH, PADDLE_HEIGHT = 10, 60
FPS = 60
WHITE = ( 255, 255, 255 )
```

These lines **def**ine constants for the game window dimensions, ball and paddle sizes, frames per second, and the color white in RGB format .

Python Code

```
# Create screen
screen = pygame . display . set_mode (( WIDTH, HEIGHT ))
pygame . display . set_caption (" Pong Game ")
```

These lines create the game window with the specified width and height and set the window caption .

Python Code

```
# Create paddles and ball
player_paddle = pygame . Rect ( 50, HEIGHT // 2 - PADDLE_HEIGHT // 2, PADDLE_WIDTH, PADDLE_HEIGHT )
opponent_paddle = pygame . Rect ( WIDTH - 50 - PADDLE_WIDTH, HEIGHT // 2 - PADDLE_HEIGHT // 2, PADDLE_WIDTH, PADDLE_HEIGHT )
```

ball = pygame . Rect (WIDTH // 2 - BALL_RADIUS // 2, HEIGHT // 2 - BALL_RADIUS // 2, BALL_RADIUS, BALL_RADIUS)

These lines create rectangles representing the player's paddle, opponent's paddle, and the ball . The initial positions and sizes are specified .

Python Code

```
# Initialize velocities
ball_speed_x = 3 * random . choice ([ 1, - 1 ])
ball_speed_y = 3 * random . choice ([ 1, - 1 ])
player_speed = 0
opponent_speed = 2 # Adjust opponent's paddle speed
```

Velocity variables for the ball and paddles are initialized . The ball starts with a random speed in both x and y directions . The player's and opponent's paddle speeds are also initialized .

Python Code

```
# Initialize score
player_score = 0
opponent_score = 0
```

Initial scores for the player and opponent are set to zero .

Python Code

```
# Function to reset the ball's position
def reset_ball ():
    ball . x = WIDTH // 2 - BALL_RADIUS // 2
    ball . y = HEIGHT // 2 - BALL_RADIUS // 2
```

This function resets the ball's position to the center of the screen .

Python Code

```
# Game loop
clock = pygame . time . Clock ()
while True :
    for event in pygame . event . get ():
        # Event handling code
```

The game loop begins here . It continuously checks for events, such as user input or quitting the game .

Python Code

```
# Move paddles and ball
player_paddle . y += player_speed
player_paddle . y = max ( 0, min ( player_paddle . y, HEIGHT - PADDLE_HEIGHT ))
```

```
ball . x += ball_speed_x
ball . y += ball_speed_y
```

These lines update the positions of the player's paddle and the ball based on their respective velocities . The player's paddle movement is restricted to stay within the game window .

Python Code

```
# Ball collisions with walls
if ball . top < = 0 or ball . bottom > = HEIGHT :
    ball_speed_y = - ball_speed_y
```

This checks if the ball hits the top or bottom walls, reversing its vertical direction if a collision occurs .

Python Code

```
# Ball collisions with paddles
if ball . colliderect ( player_paddle ):
    ball_speed_x = abs ( ball_speed_x )
elif ball . colliderect ( opponent_paddle ):
    ball_speed_x = - abs ( ball_speed_x )
```

These lines check for collisions between the ball and the player's or opponent's paddles . If a collision occurs, the ball's horizontal direction is reversed .

Python Code

```
# Check if the ball passed the paddles
if ball . left < = 0 :
    opponent_score += 1
    reset_ball ()
elif ball . right > = WIDTH :
    player_score += 1
    reset_ball ()
```

These lines check if the ball has passed the left or right sides of the window . If so, the opponent or player scores are increased, and the ball is reset to the center .

Python Code

```
# Opponent AI
if opponent_paddle . centery < ball . centery :
    opponent_paddle . y += min ( opponent_speed, ball . centery - opponent_paddle . centery )
elif opponent_paddle . centery > ball . centery :
    opponent_paddle . y -= min ( opponent_speed, opponent_paddle . centery - ball . centery )
```

This implements a simple AI for the opponent paddle, making it follow the ball vertically .

Python Code

Draw everything

Draw scores

Update the display

Cap the frame rate

These lines handle the drawing of paddles, ball, scores, and the middle line . The display is updated, and the frame rate is capped to maintain a consistent speed .

This concludes the explanation of the Pong game code . It covers the setup, game loop, user input, ball and paddle movement, collision detection, scoring, and opponent AI .

How To Play Pong Game

To play the Pong game, you control a paddle on one side of the screen, and your goal is to hit the ball past your opponent's paddle on the opposite side . Here's a step - by - step guide on how to play :

1. **Launch the Game :**
    - Run the Python script with the provided Pong code .

- The game window will appear, displaying the paddles, the ball, and the scores .

2. **Controls :**

- Use the **up arrow key** to move your paddle up .
- Use the **down arrow key** to move your paddle down .
- Press the **'R' key** to restart the game if needed .

3. **Game Objective :**

- Your objective is to prevent the ball from passing your paddle while trying to hit the ball past your opponent's paddle .

4. **Paddle Movement :**

- Move your paddle up and down to position it for hitting the ball .
- Be strategic in your movements to intercept the ball and send it towards your opponent's side .

5. **Ball Movement :**

- The ball will bounce off the top and bottom walls, as well as the paddles .
- If the ball passes your opponent's paddle on the left side or your paddle on the right side, your opponent or you score a point, respectively .

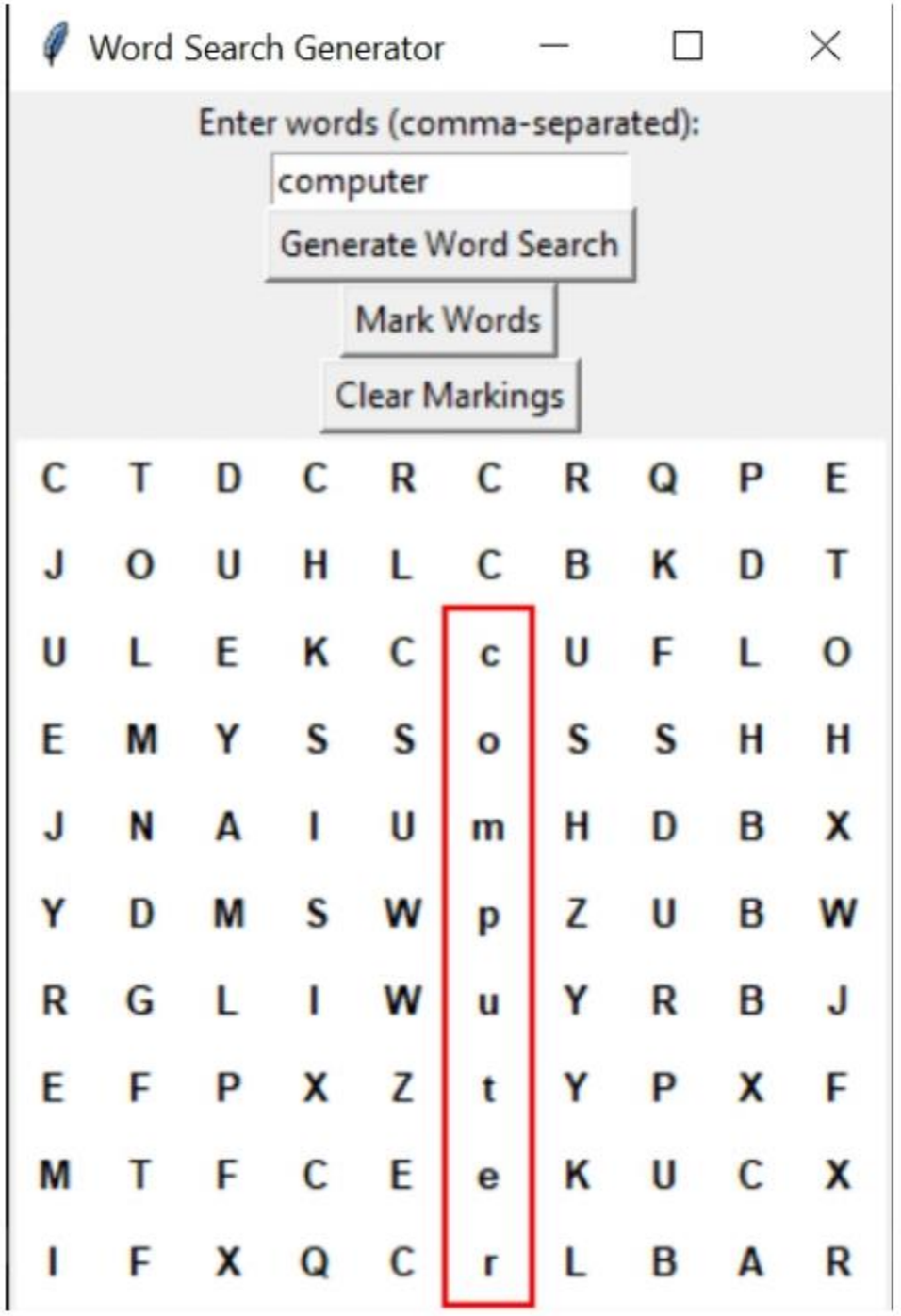
6. **Scoring :**

- The scores are displayed at the top of the screen .
- If the ball passes your opponent's paddle, your opponent scores a point .
- If the ball passes your paddle, you score a point .

7. **Game Over :**
   - The game continues in**def**initely until you decide to close the window or exit the game manually .
   - You can also press the **'R' key** to restart the game at any time .
8. **Opponent AI :**
   - The opponent paddle has its own AI to follow the ball vertically, making the game more challenging .
9. **Enjoy the Game :**
   - Have fun playing Pong ! Sharpen your reflexes and aim to outscore your opponent .

Remember, Pong is a classic and straightforward game, making it easy to pick up and play . The more you play, the better you'll become at predicting the ball's movements and scoring against your opponent .

15. Word Search Generator Game



```
import tkinter as tk
import random
```

```
class WordSearchGenerator:
  def __init__(self, root):
    self.root = root
    self.root.title("Word Search Generator")

    self.word_list_label = tk.Label(
      root, text="Enter words (comma-separated):")
    self.word_list_label.pack()

    self.word_list_entry = tk.Entry(root)
    self.word_list_entry.pack()

    self.generate_button = tk.Button(
      root, text="Generate Word Search", command=self.generate_word_search)
    self.generate_button.pack()

    self.mark_button = tk.Button(
      root, text="Mark Words", command=self.mark_words)
    self.mark_button.pack()

    self.clear_button = tk.Button(
```

```
        root, text="Clear Markings", command=self.clear_markings)
    self.clear_button.pack()

    self.word_search_canvas = tk.Canvas(
        root, width=300, height=300, bg="white")
    self.word_search_canvas.pack()

def clear_markings(self):
    if hasattr(self, 'word_search_canvas'):
        self.word_search_canvas.delete("markings")

def generate_word_search(self):
    self.clear_canvas()
    words = self.word_list_entry.get().split(',')
    self.word_search = self.create_word_search(words)
    self.display_word_search(self.word_search)

def mark_words(self):
    if hasattr(self, 'word_search'):
        words_to_mark = self.word_list_entry.get().split(',')
        for i in range(len(self.word_search)):
            for j in range(len(self.word_search[i])):
                for word in words_to_mark:
                    # Check horizontally
```

```
            if self.word_search[i][j:j + len(word)] == list(word):
                self.mark_rectangle(i, j, len(word), "horizontal")

            # Check vertically
            if i + len(word) <= len(self.word_search) and all(self.word_search[i + k][j] == word[k] for k in range(len(word))):
                self.mark_rectangle(i, j, len(word), "vertical")

            # Check diagonally (top-left to bottom-right)
            if i + len(word) <= len(self.word_search) and j + len(word) <= len(self.word_search[i]) and all(self.word_search[i + k][j + k] == word[k] for k in range(len(word))):
                self.mark_diagonal(i, j, len(word), "diagonal1")

            # Check diagonally (top-right to bottom-left)
            if i + len(word) <= len(self.word_search) and j - len(word) >= -1 and all(self.word_search[i + k][j - k] == word[k] for k in range(len(word))):
                self.mark_diagonal(i, j, len(word), "diagonal2")

    def mark_diagonal(self, start_row, start_col, length, direction):
        cell_size = 30

        if direction == "diagonal1":
            self.word_search_canvas.create_rectangle(start_col * cell_size, start_row * cell_size,
                                                     (start_col + length) *
                                                     cell_size, (start_row +
```

```
                                length) * cell_size,
                            outline="red", width=2, tags="markings")
    elif direction == "diagonal2":
        self.word_search_canvas.create_rectangle((start_col - length + 1) * cell_size, start_row * cell_size,
                            (start_col + 1) *
                            cell_size, (start_row +
                                length) * cell_size,
                            outline="red", width=2, tags="markings")

def mark_rectangle(self, start_row, start_col, length, direction):
    cell_size = 30
    if direction == "horizontal":
        self.word_search_canvas.create_rectangle(start_col * cell_size, start_row * cell_size,
                            (start_col + length) *
                            cell_size, (start_row +
                                1) * cell_size,
                            outline="red", width=2, tags="markings")
    elif direction == "vertical":
        self.word_search_canvas.create_rectangle(start_col * cell_size, start_row * cell_size,
                            (start_col + 1) *
                            cell_size, (start_row +
                                length) * cell_size,
                            outline="red", width=2, tags="markings")
    elif direction == "diagonal":
```

```
        self.word_search_canvas.create_rectangle(start_col * cell_size, start_row * cell_size,
                                 (start_col + length) *
                                 cell_size, (start_row +
                                         length) * cell_size,
                                 outline="red", width=2, tags="markings")

    def clear_canvas(self):
        self.word_search_canvas.delete("all")

    def display_word_search(self, word_search):
        cell_size = 30
        for i in range(len(word_search)):
            for j in range(len(word_search[i])):
                self.word_search_canvas.create_text(j * cell_size + cell_size // 2, i * cell_size + cell_size // 2,
                                 text=word_search[i][j], font=("Helvetica", 10, "bold"))

    def create_word_search(self, words):
        word_search_size = 10
        word_search = [[' ' for _ in range(word_search_size)]
                       for _ in range(word_search_size)]

        for word in words:
            placed = False
            attempts = 0
```

```
    while not placed and attempts < 100:
      direction = random.choice(
        ['horizontal', 'vertical', 'diagonal'])
      start_row = random.randint(0, len(word_search) - 1)
      start_col = random.randint(0, len(word_search[0]) - 1)

      if direction == 'horizontal' and start_col + len(word) <= word_search_size:
        for i in range(len(word)):
          word_search[start_row][start_col + i] = word[i]
        placed = True

      elif direction == 'vertical' and start_row + len(word) <= word_search_size:
        for i in range(len(word)):
          word_search[start_row + i][start_col] = word[i]
        placed = True

          elif direction == 'diagonal' and start_row + len(word) <= word_search_size and start_col + len(word) <=
word_search_size:
        for i in range(len(word)):
          word_search[start_row + i][start_col + i] = word[i]
        placed = True

      attempts += 1
```

```
    # Fill in the remaining spaces with random letters
    for i in range(word_search_size):
        for j in range(word_search_size):
            if word_search[i][j] == ' ':
                word_search[i][j] = chr(random.randint(65, 90))

    return word_search
```

```
if __name__ == "__main__":
    root = tk.Tk()
    app = WordSearchGenerator(root)
    root.mainloop()
```

let's go through the code line by line to understand its functionality :

Python Code

```
import tkinter as tk
import random
```

- Import the tkinter library for GUI and random for generating random numbers .

Python Code

```
class WordSearchGenerator :
```

```
def __init__ ( self, root ):
    self . root  =  root
    self . root . title (" Word Search Generator ")
```

- **Def**ine a class WordSearchGenerator with an __init__ method . It takes the root as a parameter, representing the Tkinter root window . Sets the title of the root window to " Word Search Generator ".

Python Code

```
    self . word_list_label  =  tk . Label (
            root, text =" Enter words  ( comma - separated ):")
    self . word_list_label . pack ()
```

- Create a Tkinter label for instructing the user to enter words . The label is added to the root window .

Python Code

```
    self . word_list_entry  =  tk . Entry ( root )
    self . word_list_entry . pack ()
```

- Create a Tkinter entry widget for users to input words (comma - separated). It is added to the root window .

Python Code

```
    self . generate_button  =  tk . Button (
```

```
        root, text =" Generate Word Search " , command = self . generate_word_search )
    self . generate_button . pack ()
```

- Create a button labeled " Generate Word Search " with a command to call the generate_word_search method when clicked . The button is added to the root window .

Python Code

```
    self . mark_button = tk . Button (
        root, text =" Mark Words " , command = self . mark_words )
    self . mark_button . pack ()
```

- Create a button labeled " Mark Words " with a command to call the mark_words method when clicked . The button is added to the root window .

Python Code

```
    self . clear_button = tk . Button (
        root, text =" Clear Markings " , command = self . clear_markings )
    self . clear_button . pack ()
```

- Create a button labeled " Clear Markings " with a command to call the clear_markings method when clicked . The button is added to the root window .

Python Code

```
    self . word_search_canvas = tk . Canvas (
```

```
            root, width = 300, height = 300, bg =" white ")
        self . word_search_canvas . pack ()
```

- Create a Tkinter canvas for displaying the word search grid . It is given a size of 300x300 pixels and a white background . The canvas is added to the root window .

Python Code

```
    def clear_markings ( self ):
        if hasattr ( self, 'word_search_canvas' ):
            self . word_search_canvas . delete (" markings ")
```

- **Def**ine a method clear_markings to clear any marked rectangles on the canvas . It checks if the canvas attribute exists before attempting to delete markings .

Python Code

```
    def generate_word_search ( self ):
        self . clear_canvas ()
        words = self . word_list_entry . get (). split ( ',' )
        self . word_search = self . create_word_search ( words )
        self . display_word_search ( self . word_search )
```

- **Def**ine a method generate_word_search to generate a new word search grid . It first clears the canvas, then retrieves the words from the entry widget . It generates a word search using the create_word_search method and displays it using display_word_search .

Python Code

```
def mark_words ( self ):
    if hasattr ( self, 'word_search' ):
        words_to_mark = self . word_list_entry . get (). split ( ',' )
        for i in range ( len ( self . word_search )):
            for j in range ( len ( self . word_search [ i ])):
                for word in words_to_mark :
                    # Check horizontally
                    if self . word_search [ i ][ j : j + len ( word )] == list ( word ):
                        self . mark_rectangle ( i, j, len ( word ) , " horizontal ")

                    # Check vertically
                    if i + len ( word ) < = len ( self . word_search ) and
all ( self . word_search [ i + k ][ j ] == word [ k ] for k in range ( len ( word ))):
                        self . mark_rectangle ( i, j, len ( word ) , " vertical ")
```

```
                # Check diagonally (top-left to bottom-right)
                if i + len(word) <= len(self.word_search) and j + len(word)
<= len(self.word_search[i]) and all(self.word_search[i + k][j + k] == word[k]
for k in range(len(word))):
                    self.mark_diagonal(i, j, len(word), "diagonal1")

                # Check diagonally (top-right to bottom-left)
                if i + len(word) <= len(self.word_search) and j - len(word)
>= -1 and all(self.word_search[i + k][j - k] == word[k] for k in range(len(word))):
                    self.mark_diagonal(i, j, len(word), "diagonal2")
```

- **Def**ine a method `mark_words` to mark the specified words on the word search grid. It checks for each word in different directions (horizontal, vertical, and diagonals) and marks rectangles accordingly using `mark_rectangle` and `mark_diagonal` methods.

Python Code

```
    def mark_diagonal ( self, start_row, start_col, length, direction ):
        cell_size  =  30

        if direction  == " diagonal1 ":
                self . word_search_canvas . create_rectangle ( start_col  *  cell_size, start_row  *
cell_size,
                                                                ( start_col  +  length ) *
                                                                cell_size,  ( start_row  +
                                                                        length ) *  cell_size,
                                                                outline =" red " ,                width = 2,
tags =" markings ")
        elif direction  == " diagonal2 ":
                self . word_search_canvas . create_rectangle (( start_col   -   length   +   1 )   *
cell_size, start_row  *  cell_size,
                                                                ( start_col  +  1 ) *
                                                                cell_size,  ( start_row  +
                                                                        length ) *  cell_size,
```

```
                                                    outline =" red " ,          width = 2,
tags =" markings ")
```

- **Def**ine a method mark_diagonal to mark a diagonal rectangle on the canvas . The direction determines the diagonal orientation .

Python Code

```
  def mark_rectangle ( self, start_row, start_col, length, direction ):
    cell_size  =  30
    if direction  == " horizontal ":
            self . word_search_canvas . create_rectangle ( start_col  *  cell_size, start_row  *
cell_size,
                                                    ( start_col  +  length ) *
                                                    cell_size,  ( start_row  +
                                                            1 ) *  cell_size,
                                                    outline =" red " ,          width = 2,
tags =" markings ")
    elif direction  == " vertical ":
            self . word_search_canvas . create_rectangle ( start_col  *  cell_size, start_row  *
cell_size,
```

```
                                                    ( start_col + 1 )*
                                                    cell_size, ( start_row +
                                                                length )* cell_size,
                                                    outline =" red " ,          width = 2,
tags =" markings ")
    elif direction == " diagonal ":
            self . word_search_canvas . create_rectangle ( start_col * cell_size, start_row *
cell_size,
                                                    ( start_col + length )*
                                                    cell_size, ( start_row +
                                                                length )* cell_size,
                                                    outline =" red " ,          width = 2,
tags =" markings ")
```

- **Def**ine a method mark_rectangle to mark a rectangle on the canvas . The direction parameter determines if it's horizontal, vertical, or diagonal .

Python Code

```
  def clear_canvas ( self ):
    self . word_search_canvas . delete (" all ")
```

- **Def**ine a method clear_canvas to clear the entire canvas .

Python Code

```
    def display_word_search ( self, word_search ):
        cell_size = 30
        for i in range ( len ( word_search )):
                for j in range ( len ( word_search [ i ])):
                    self . word_search_canvas . create_text ( j * cell_size + cell_size // 2, i *
cell_size + cell_size // 2,
                                                                text = word_search [ i ][ j ] ,
font =(" Helvetica " , 10, " bold "))
```

- **Def**ine a method display_word_search to display the generated word search on the canvas . It uses create_text to place each character in the appropriate cell .

Python Code

```
    def create_word_search ( self, words ):
        word_search_size = 10
        word_search = [[ '' for _ in range ( word_search_size )]
                            for _ in range ( word_search_size )]
```

```
    for word in words :
            placed = False
            attempts = 0

            while not placed and attempts < 100 :
                direction = random . choice (
                    [ 'horizontal', 'vertical', 'diagonal' ])
                start_row = random . randint ( 0, len ( word_search ) - 1 )
                start_col = random . randint ( 0, len ( word_search [ 0 ]) - 1 )

                if direction == 'horizontal' and start_col + len ( word ) <=
word_search_size :
                    for i in range ( len ( word )):
                        word_search [ start_row ][ start_col + i ] = word [ i ]
                    placed = True

                elif direction == 'vertical' and start_row + len ( word ) <=
word_search_size :
                    for i in range ( len ( word )):
                        word_search [ start_row + i ][ start_col ] = word [ i ]
```

```
                placed = True

            elif direction == 'diagonal' and start_row + len(word) <=
word_search_size and start_col + len(word) <= word_search_size:
                for i in range(len(word)):
                    word_search[start_row + i][start_col + i] = word[i]
                placed = True

            attempts += 1

    # Fill in the remaining spaces with random letters
    for i in range(word_search_size):
        for j in range(word_search_size):
            if word_search[i][j] == '':
                word_search[i][j] = chr(random.randint(65, 90))

    return word_search
```

- **Def**ine a method create_word_search to generate a word search grid based on the input words . It randomly places the words in different directions and fills the remaining spaces with random letters .

Python Code

```
if __name__ == " __main__ ":
    root = tk . Tk ()
    app = WordSearchGenerator ( root )
    root . mainloop ()
```

- If the script is run as the main program, create a Tkinter root window and instantiate the WordSearchGenerator class, starting the Tkinter event loop with root . mainloop () .

How To Play Word Search Generator

To play the Word Search Generator, follow these steps :

1. **Launch the Application :**
    - Run the Python script containing the Word Search Generator code .
    - The graphical user interface (GUI) will appear, featuring an entry field, buttons, and a canvas .
2. **Enter Words :**
    - In the " Enter words (comma - separated):" entry field, type the words you want to search for in the word search grid . Separate the words with commas .
3. **Generate Word Search :**

- Click the " Generate Word Search " button . This will create a word search grid using the entered words and display it on the canvas .

4. **Mark Words :**

- After generating the word search, you can mark specific words on the grid . Enter the words you want to mark in the entry field again .
- Click the " Mark Words " button . The application will search for the entered words horizontally, vertically, and diagonally on the grid . It will mark the found words with red rectangles on the canvas .

5. **Clear Markings :**

- If you want to clear the marked rectangles from the canvas, click the " Clear Markings " button .

6. **Explore the Word Search :**

- Explore the word search grid visually . The marked words will be highlighted, making them easier to locate .

7. **Additional Features :**

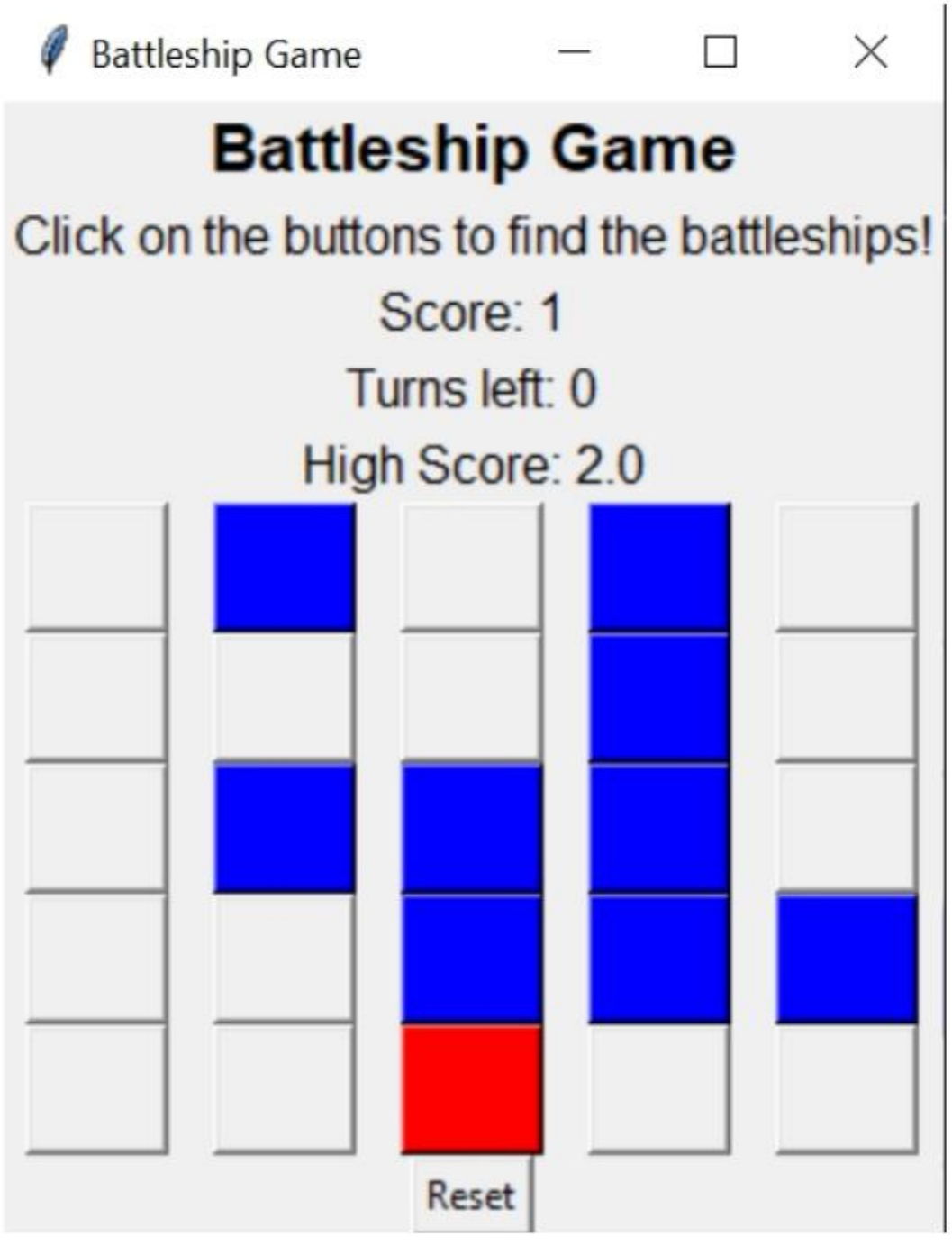
- You can modify the word list, generate a new word search, mark different words, and clear markings as many times as you want .

8. **Close the Application :**

- Close the application window when you're finished playing .

Enjoy playing the Word Search Generator, and have fun finding the hidden words in the generated grids !

16. Battleship Game



```
import tkinter as tk
from random import randint
import time
```

```
class BattleshipGame:
  def __init__(self, root):
    self.root = root
    self.root.title("Battleship Game")

    self.board_size = 5
    self.ship_size = 3
    self.max_turns = 10
    self.turns_left = self.max_turns
    self.score = 0
    self.board = [[0] * self.board_size for _ in range(self.board_size)]
    self.ships = []

    self.load_high_score()
    self.create_info_panel()
    self.create_board()
    self.place_ships()

    self.start_time = time.time()
```

```
    # Add a Reset button
    reset_button = tk.Button(
        self.root, text="Reset", command=self.reset_game)
    reset_button.grid(row=self.board_size + 6, column=0,
                      columnspan=self.board_size)

def load_high_score(self):
    try:
        with open("high_score.txt", "r") as file:
            high_score_str = file.read().strip()
            if high_score_str.lower() == "inf":
                self.high_score = float('inf')
            else:
                self.high_score = float(high_score_str)
    except FileNotFoundError:
        self.high_score = float('inf')

def save_high_score(self):
    with open("high_score.txt", "w") as file:
        if self.high_score == float('inf'):
            file.write("inf")
        else:
            file.write(f"{self.high_score:.2f}")
```

```
  def create_info_panel(self):
    info_label = tk.Label(
      self.root, text="Battleship Game", font=("Helvetica", 16, "bold"))
    info_label.grid(row=0, column=0, columnspan=self.board_size)

    instruction_label = tk.Label(
      self.root, text="Click on the buttons to find the battleships!", font=("Helvetica", 12))
    instruction_label.grid(row=1, column=0, columnspan=self.board_size)

    score_label = tk.Label(
      self.root, text=f"Score: {self.score}", font=("Helvetica", 12))
    score_label.grid(row=2, column=0, columnspan=self.board_size)

    turns_label = tk.Label(
      self.root, text=f"Turns left: {self.turns_left}", font=("Helvetica", 12))
    turns_label.grid(row=3, column=0, columnspan=self.board_size)

    high_score_label = tk.Label(
     self.root, text=f"High Score: {'inf' if self.high_score == float('inf') else round(self.high_score, 2)}", font=("Helvetica",
12))
    high_score_label.grid(row=4, column=0, columnspan=self.board_size)

  def create_board(self):
    for i in range(self.board_size):
```

```
        for j in range(self.board_size):
            btn = tk.Button(self.root, text="", width=5, height=2,
                            command=lambda i=i, j=j: self.click_cell(i, j))
            btn.grid(row=i + 5, column=j)

    def place_ships(self):
        for _ in range(self.ship_size):
            ship_row = randint(0, self.board_size - 1)
            ship_col = randint(0, self.board_size - 1)
            while self.board[ship_row][ship_col] == 1:
                ship_row = randint(0, self.board_size - 1)
                ship_col = randint(0, self.board_size - 1)
            self.ships.append((ship_row, ship_col))
            self.board[ship_row][ship_col] = 1

    def update_info_panel(self):
        self.root.grid_slaves(row=2, column=0)[0].config(
            text=f"Score: {self.score}")
        self.root.grid_slaves(row=3, column=0)[0].config(
            text=f"Turns left: {self.turns_left}")
        self.root.grid_slaves(row=4, column=0)[0].config(
            text=f"High Score: {'inf' if self.high_score == float('inf') else round(self.high_score, 2)}")

    def display_message(self, message):
```

```
    message_label = tk.Label(
        self.root, text=message, font=("Helvetica", 14, "bold"))
    message_label.grid(row=1, column=0, columnspan=self.board_size)
    self.root.after(2000, message_label.destroy)

def reset_game(self):
    if self.score > 0:
        # Save the current score as high score if some ships were sunk
        if self.score < self.high_score:
            self.high_score = self.score
            self.save_high_score()
            self.root.grid_slaves(row=4, column=0)[0].config(
                text=f"High Score: {round(self.high_score, 2)}")

    self.score = 0
    self.turns_left = self.max_turns
    self.ships = []
    self.board = [[0] * self.board_size for _ in range(self.board_size)]
    self.update_info_panel()

    default_bg_color = self.root.cget("bg")

    for i in range(self.board_size):
        for j in range(self.board_size):
```

```
        btn = self.root.grid_slaves(row=i + 5, column=j)[0]
        btn.config(state=tk.NORMAL, bg=default_bg_color, text="")

    if self.turns_left > 0:
      self.place_ships()

    self.start_time = time.time()

  def click_cell(self, row, col):
    if self.turns_left > 0:
      self.root.grid_slaves(
        row=row + 5, column=col)[0].config(state=tk.DISABLED)

      if (row, col) in self.ships:
        self.score += 1
        self.root.grid_slaves(
          row=row + 5, column=col)[0].config(bg="red")
      else:
        self.root.grid_slaves(
          row=row + 5, column=col)[0].config(bg="blue")

      self.turns_left -= 1
      self.update_info_panel()
```

```
        if self.score == self.ship_size:
            elapsed_time = round(time.time() - self.start_time, 2)
            self.display_message(
                f"Congratulations! You sunk all the battleships in {elapsed_time} seconds.")
            if elapsed_time < self.high_score:
                self.high_score = elapsed_time
                self.save_high_score()
                self.root.grid_slaves(row=4, column=0)[0].config(
                    text=f"High Score: {'inf' if self.high_score == float('inf') else round(self.high_score, 2)}")
            self.reset_game()  # Reset only when all ships are sunk
        elif self.turns_left == 0:
            self.display_message("Game Over. You ran out of turns.")
            if self.score > 0 and self.score < self.ship_size:
                # If some ships were sunk, save the current score as high score
                if self.score > self.high_score:
                    self.high_score = self.score
                    self.save_high_score()
                    self.root.grid_slaves(row=4, column=0)[0].config(
                        text=f"High Score: {round(self.high_score, 2)}")
```

```
if __name__ == "__main__":
    root = tk.Tk()
    game = BattleshipGame(root)
```

```
root.mainloop()
```

let's go through the code line by line to understand each part :

Python Code

```
import tkinter as tk
from random import randint
import time
```

This section imports the necessary modules for creating a graphical user interface (GUI) with Tkinter, generating random numbers with randint , and handling time - related functions with time .

Python Code

```
class BattleshipGame :
    def __init__ ( self, root ):
```

Here, a class BattleshipGame is **def**ined . The __init__ method serves as the constructor for the class . It takes an argument root , which is the Tkinter root window .

Python Code

```
        self . root = root
        self . root . title (" Battleship Game ")
```

This initializes the root attribute with the provided root window and sets the window title to " Battleship Game ."

Python Code

```
self . board_size = 5
self . ship_size = 3
self . max_turns = 10
self . turns_left = self . max_turns
self . score = 0
```

These lines **def**ine several game - related parameters, such as the board size, ship size, maximum number of turns, current turns left, and the player's score .

Python Code

```
self . board = [[ 0 ] * self . board_size for _ in range ( self . board_size )]
self . ships = []
```

Here, the board is a 2D list initialized with zeros, representing the game board . The ships list will store the coordinates of the ships on the board .

Python Code

```
self . load_high_score ()
self . create_info_panel ()
```

```
self . create_board ()
self . place_ships ()
```

These lines call methods to load the high score from a file, create the information panel in the GUI, create the game board buttons, and place the ships randomly on the board .

Python Code

```
self . start_time = time . time ()
```

This records the starting time of the game using the time module .

Python Code

```
reset_button = tk . Button (
        self . root, text =" Reset " , command = self . reset_game )
reset_button . grid ( row = self . board_size + 6, column = 0,
                      columnspan = self . board_size )
```

This creates a " Reset " button in the GUI, which calls the reset_game method when clicked .

Python Code

```
def load_high_score ( self ):
```

This method loads the high score from a file (" high_score . txt ") and initializes the high_score attribute with the retrieved value .

Python Code

```
def save_high_score ( self ):
```

This method saves the current high score to the same file .

Python Code

```
def create_info_panel ( self ):
```

This method creates labels in the GUI to display information such as the game title, instructions, score, turns left, and high score .

Python Code

```
def create_board ( self ):
```

This method creates buttons in the GUI to represent the game board . Each button is associated with the click_cell method when clicked .

Python Code

```
def place_ships ( self ):
```

This method randomly places ships on the game board and updates the ships list .

Python Code

```
def update_info_panel ( self ):
```

This method updates the information panel in the GUI to reflect the current score, turns left, and high score .

Python Code

```
def display_message ( self, message ):
```

This method displays a temporary message on the GUI for 2 seconds . It is used for congratulatory and game - over messages .

Python Code

```
def reset_game ( self ):
```

This method resets the game, saving the current score as the high score if it's better . It also resets the game board, score, turns left, and ships .

Python Code

```
def click_cell ( self, row, col ):
```

This method is called when a game board button is clicked . It disables the button, updates the score, turns left, and checks for game completion conditions .

Python Code

```
if __name__ == " __main__ ":
```

This block checks if the script is the main module, and if so, it creates an instance of the BattleshipGame class and starts the Tkinter main loop .

Python Code

```
root = tk . Tk ()
game = BattleshipGame ( root )
root . mainloop ()
```

Here, a Tkinter root window is created, and the BattleshipGame instance is instantiated with this root window . The Tkinter main loop (root . mainloop ()) is then started, allowing the GUI to be displayed and interacted with .

How To Play Battleship Game

1. **Run the Script :** Execute the Python script that contains the Battleship game code . This will open a graphical user interface (GUI) window .
2. **Game Layout :**
    - The game board consists of a grid of buttons .
    - The title " Battleship Game " is displayed at the top of the window .
    - Below the title, there are instructions guiding you to click on the buttons to find the battleships .

- The score, turns left, and high score are displayed in the information panel .

3. **Click Buttons :**

- To uncover battleships, click on the buttons in the grid .
- Each button represents a cell on the game board .

4. **Gameplay :**

- The game starts with battleships randomly placed on the board .
- Clicking a button reveals whether a battleship is present in that cell .
- If you hit a battleship (click on a cell containing a battleship) , your score increases .
- If you miss a battleship, the clicked button turns blue .
- You have a limited number of turns to find and sink all the battleships .

5. **Game Over :**

- The game ends when you run out of turns .
- A message will be displayed, indicating that the game is over, and you've exhausted all turns .

6. **Congratulations :**

- If you successfully sink all the battleships within the allowed turns, a congratulatory message will be displayed .

- The elapsed time to complete the game is also shown in seconds .

7. **Reset the Game :**

- After the game concludes, you can reset the game by clicking the " Reset " button .
- If you achieved a high score during the game, it will be saved .

8. **High Score :**

- The high score is displayed in the information panel .
- The goal is to complete the game in the shortest time possible to achieve a lower high score .

9. **Closing the Game :**

- You can close the game window at any time .

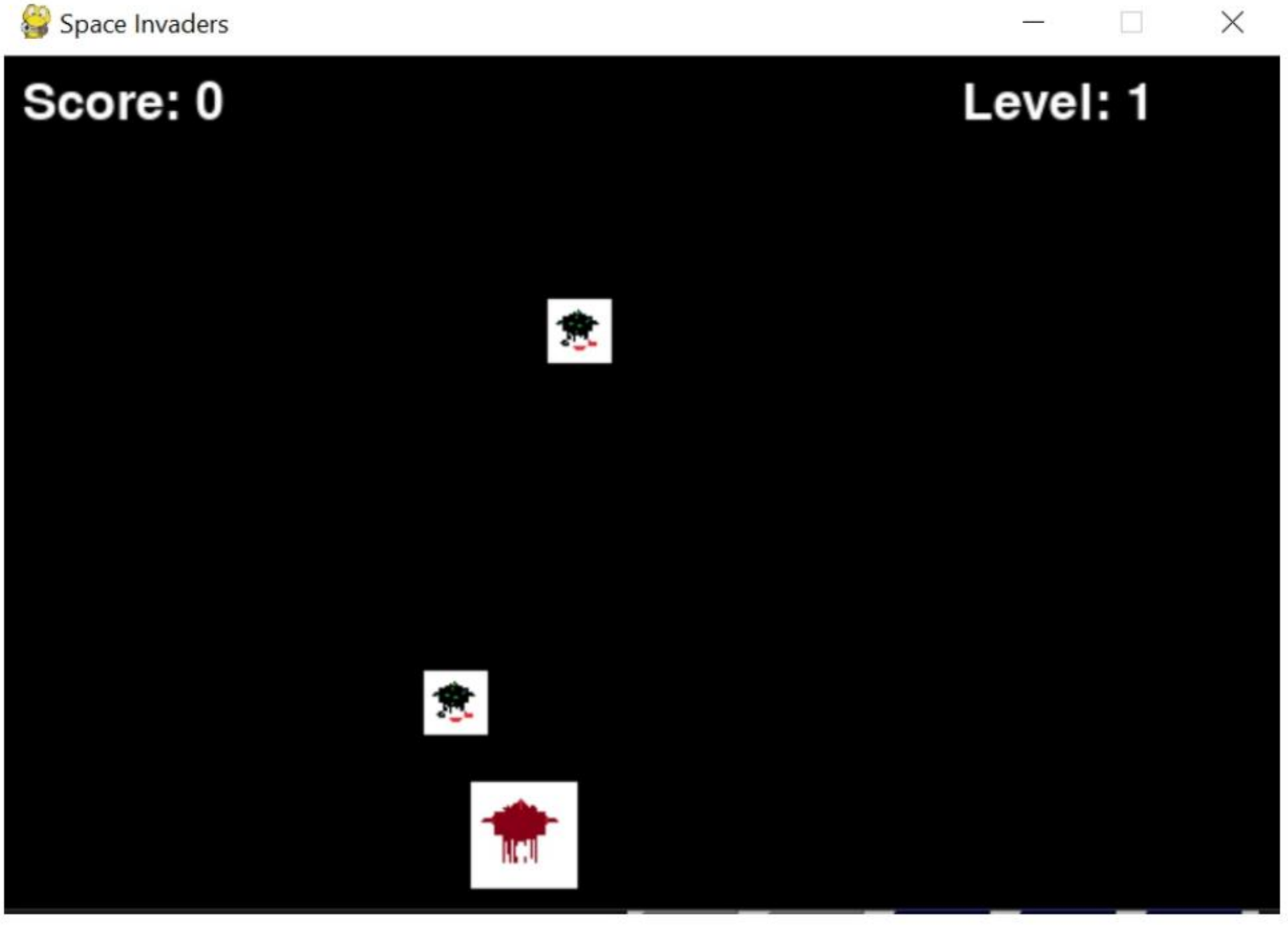
10. **Restarting the Game :**

- To play again, run the script or restart the Python program .

Enjoy playing Battleship and try to beat your high score by sinking all the battleships in the shortest time !

---

17. Space Invader Game



```
import pygame
import sys
import random

# Initialize Pygame
pygame.init()

# Constants
WIDTH, HEIGHT = 600, 400
FPS = 30
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)

# Player
player_size = 50
player_speed = 5

# Enemy
enemy_size = 30
enemy_speed = 2
initial_enemy_spawn_rate = 25
min_enemy_spawn_rate = 5  # Minimum spawn rate

# Bullet
```

```
bullet_speed = 7

# Initialize the screen
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Space Invaders")

# Load images
player_image = pygame.image.load("player.png")
enemy_image = pygame.image.load("enemy.png")
bullet_image = pygame.image.load("bullet.png")

# Load sounds
explosion_sound = pygame.mixer.Sound("explosion.wav")
shooting_sound = pygame.mixer.Sound("shooting.wav")

# Clock for controlling the frame rate
clock = pygame.time.Clock()

# Player class
```

```
class Player(pygame.sprite.Sprite):
  def __init__(self):
    super().__init__()
    self.image = pygame.transform.scale(
```

```
        player_image, (player_size, player_size))
    self.rect = self.image.get_rect()
    self.rect.centerx = WIDTH // 2
    self.rect.bottom = HEIGHT - 10

  def update(self):
    keys = pygame.key.get_pressed()
    if keys[pygame.K_LEFT] and self.rect.left > 0:
      self.rect.x -= player_speed
    if keys[pygame.K_RIGHT] and self.rect.right < WIDTH:
      self.rect.x += player_speed

# Enemy class
```

```
class Enemy(pygame.sprite.Sprite):
  def __init__(self):
    super().__init__()
    self.image = pygame.transform.scale(
      enemy_image, (enemy_size, enemy_size))
    self.rect = self.image.get_rect()
    self.rect.x = random.randint(0, WIDTH - enemy_size)
    self.rect.y = random.randint(-HEIGHT, 0)

  def update(self):
```

```
        self.rect.y += enemy_speed
        if self.rect.top > HEIGHT:
            self.rect.x = random.randint(0, WIDTH - enemy_size)
            self.rect.y = random.randint(-HEIGHT, 0)

# Bullet class
```

```
class Bullet(pygame.sprite.Sprite):
    def __init__(self, x, y):
        super().__init__()
        self.image = pygame.transform.scale(bullet_image, (10, 20))
        self.rect = self.image.get_rect()
        self.rect.centerx = x
        self.rect.bottom = y

    def update(self):
        self.rect.y -= bullet_speed
        if self.rect.bottom < 0:
            self.kill()
```

```
# Create sprite groups
all_sprites = pygame.sprite.Group()
enemies = pygame.sprite.Group()
```

```
bullets = pygame.sprite.Group()

# Create player
player = Player()
all_sprites.add(player)

# Scoring and level variables
score = 0
level = 1
font = pygame.font.Font(None, 36)

# Game over flag
game_over = False

# Main game loop
while True:
  # Event handling
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      pygame.quit()
      sys.exit()
    elif event.type == pygame.KEYDOWN:
      if not game_over:
        if event.key == pygame.K_SPACE:
```

```
            bullet = Bullet(player.rect.centerx, player.rect.top)
            all_sprites.add(bullet)
            bullets.add(bullet)
            shooting_sound.play()
        else:
            if event.key == pygame.K_r:
                # Reset game state
                all_sprites.empty()
                enemies.empty()
                bullets.empty()
                player = Player()
                all_sprites.add(player)
                score = 0
                level = 1
                enemy_speed = 2
                initial_enemy_spawn_rate = 25
                min_enemy_spawn_rate = 5
                game_over = False

    if not game_over:
        # Spawn enemies
        if random.randint(1, initial_enemy_spawn_rate) == 1:
            enemy = Enemy()
            all_sprites.add(enemy)
```

```
    enemies.add(enemy)

# Update sprites
all_sprites.update()

# Check for collisions
hits = pygame.sprite.groupcollide(enemies, bullets, True, True)
for hit in hits:
    score += 10
    explosion_sound.play()
    enemy = Enemy()
    all_sprites.add(enemy)
    enemies.add(enemy)

hits = pygame.sprite.spritecollide(player, enemies, False)
if hits:
    game_over = True

# Draw everything
screen.fill(BLACK)
all_sprites.draw(screen)

# Display score and level
score_text = font.render(f"Score: {score}", True, WHITE)
```

```
    level_text = font.render(f"Level: {level}", True, WHITE)
    screen.blit(score_text, (10, 10))
    screen.blit(level_text, (WIDTH - 150, 10))

    # Display "Game Over" message if the game is over
if game_over:
    # Draw white background
    white_rect = pygame.Rect(WIDTH // 2 - 220, HEIGHT // 2 - 40, 440, 80)
    pygame.draw.rect(screen, WHITE, white_rect)

    # Draw "Game Over" text in yellow, centered within the white rectangle
    game_over_text = font.render(
        "Game Over, Press R to Restart", True, (0, 0, 255))  # Yellow color
    text_rect = game_over_text.get_rect(center=white_rect.center)
    screen.blit(game_over_text, text_rect.topleft)
```

```
# Update display
pygame.display.flip()

# Increase difficulty with levels
if not game_over and score >= level * 100:
    level += 1
    enemy_speed += 0.001
```

```
    initial_enemy_spawn_rate -= 1  # Decrease the spawn rate
    # Ensure it doesn't go below min rate
    initial_enemy_spawn_rate = max(
        initial_enemy_spawn_rate, min_enemy_spawn_rate)

# Cap the frame rate
clock.tick(FPS)
```

let's go through the code line by line and explain what each part does :

Python Code

```
import pygame
import sys
import random
```

Here, you're importing the necessary modules : pygame for creating the game, sys for interacting with the system, and random for generating random numbers .

Python Code

```
pygame . init ()
```

This initializes Pygame .

Python Code

```
# Constants
WIDTH, HEIGHT = 600, 400
FPS = 30
WHITE = ( 255, 255, 255 )
BLACK = ( 0, 0, 0 )
```

These lines **def**ine some constants used throughout the game, such as the screen width and height, frames per second, and colors .

Python Code

```
# Player
player_size = 50
player_speed = 5
```

Defines properties for the player character, such as its size and movement speed .

Python Code

```
# Enemy
enemy_size = 30
enemy_speed = 2
initial_enemy_spawn_rate = 25
min_enemy_spawn_rate = 5 # Minimum spawn rate
```

Defines properties for the enemy characters, such as size, speed, and spawn rate .

Python Code

```
# Bullet
bullet_speed = 7
```

Defines the speed of the bullets fired by the player .

Python Code

```
# Initialize the screen
screen = pygame . display . set_mode (( WIDTH, HEIGHT ))
pygame . display . set_caption (" Space Invaders ")
```

Initializes the game window with the specified width and height and sets the window caption .

Python Code

```
# Load images
player_image = pygame . image . load (" player . png ")
enemy_image = pygame . image . load (" enemy . png ")
bullet_image = pygame . image . load (" bullet . png ")
```

Loads the images for the player, enemy, and bullet from their respective image files .

Python Code

```
# Load sounds
explosion_sound = pygame . mixer . Sound (" explosion . wav ")
shooting_sound = pygame . mixer . Sound (" shooting . wav ")
```

Loads the sound effects for explosions and shooting .

Python Code

```
# Clock for controlling the frame rate
clock = pygame . time . Clock ()
```

Creates a Clock object to control the frame rate of the game .

The next section **def**ines classes for the player, enemy, and bullet sprites, each with their respective properties and methods .

The main game loop begins with a while loop that continuously runs until the game is exited .

Inside the game loop, events such as quitting the game or pressing keys are handled .

If the game is not over, enemies are spawned randomly, sprites are updated, collisions are checked, and the screen is drawn accordingly .

If the game is over, a " Game Over " message is displayed .

Python Code

```
# Update display
pygame . display . flip ()
```

Updates the display to show the changes made during this iteration of the game loop .

Python Code

```
# Increase difficulty with levels
if not game_over and score > = level * 100 :
    level += 1
    enemy_speed += 0 . 001
    initial_enemy_spawn_rate -= 1 # Decrease the spawn rate
     # Ensure it doesn't go below min rate
    initial_enemy_spawn_rate = max (
```

initial_enemy_spawn_rate, min_enemy_spawn_rate)

Checks if the game is not over and if the score has reached a multiple of 100 . If so, increases the level, enemy speed, and decreases the spawn rate of enemies .

Python Code

Cap the frame rate
clock . tick (FPS)

Limits the frame rate to the specified FPS value, ensuring the game runs at a consistent speed across different devices .

This is a high - level overview of the code structure and functionality . Each part contributes to creating a simple space invaders game using Pygame .

How To Play Space Invader Game

To play the Space Invader game, follow these instructions :

1. **Controls :**
    - Move your player left： Press the **left arrow key .**
    - Move your player right： Press the **right arrow key .**
    - Shoot bullets： Press the **spacebar .**

2. **Objective :**

- Your goal is to shoot down the descending enemy ships (Space Invaders) while avoiding collisions with them .

3. **Player Movement :**

- Use the left and right arrow keys to move your player spaceship horizontally across the bottom of the screen .

4. **Shooting :**

- Press the spacebar to shoot bullets upward from your player spaceship .

5. **Enemy Ships :**

- Enemy ships will spawn at the top of the screen and move downward towards your player .
- Your objective is to shoot down these enemy ships before they reach the bottom of the screen .

6. **Scoring :**

- You earn points for each enemy ship you successfully shoot down .
- The score is displayed on the screen .

7. **Levels :**

- As your score increases, you will progress through levels .
- Each level may bring increased difficulty, such as faster enemy ships .

8. **Game Over :**

- The game ends if an enemy ship collides with your player spaceship .
- If you want to restart after a game over, press the **'R' key** .

9. **Restarting the Game :**

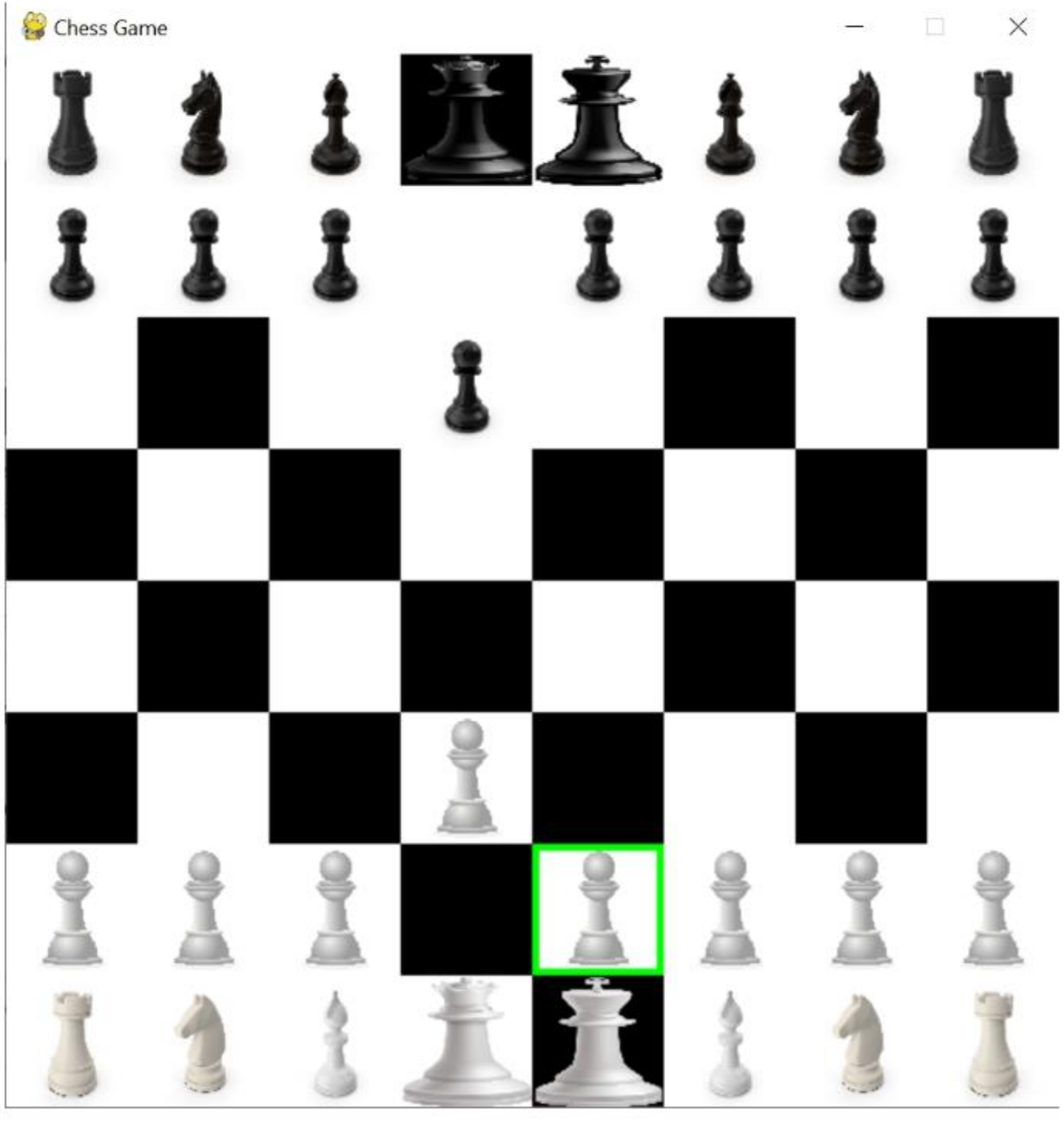
- If you see the " Game Over " message, press the **'R' key** to restart the game and play again .

10. **Enjoy and Have Fun :**

- Have fun playing Space Invaders ! Try to achieve the highest score and reach higher levels .

Remember, the game's difficulty increases as you progress through levels, so stay alert, dodge enemy fire, and aim accurately to succeed !

18. Chess Game



```
import pygame
import sys
import os

# Initialize pygame
pygame.init()

# Constants
WIDTH, HEIGHT = 600, 600
BOARD_SIZE = 8
SQUARE_SIZE = WIDTH // BOARD_SIZE
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)

# Chess board representation
chess_board = [
  ["r", "n", "b", "q", "k", "b", "n", "r"],
  ["p", "p", "p", "p", "p", "p", "p", "p"],
  ["", "", "", "", "", "", "", ""],
  ["", "", "", "", "", "", "", ""],
  ["", "", "", "", "", "", "", ""],
  ["", "", "", "", "", "", "", ""],
  ["P", "P", "P", "P", "P", "P", "P", "P"],
  ["R", "N", "B", "Q", "K", "B", "N", "R"],
```

```
]

# Initialize the pygame screen
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Chess Game")

# Load chess piece images
pieces = {}
for color in ['w', 'b']:
    for piece in ['r', 'n', 'b', 'q', 'k', 'p']:
        img_path = os.path.join("images", f"{color}{piece.lower()}.png")
        pieces[color + piece] = pygame.transform.scale(
            pygame.image.load(img_path), (SQUARE_SIZE, SQUARE_SIZE))

selected_piece = None
selected_row = None
selected_col = None
```

```
def is_valid_move(piece, start, end, board):
    row_start, col_start = start
    row_end, col_end = end

    if piece == "":
        return False  # No piece to move
```

```
  if not (0 <= row_start < 8 and 0 <= col_start < 8 and 0 <= row_end < 8 and 0 <= col_end < 8):
    return False  # Check if the move is within the board boundaries

  if (piece.islower() and row_end <= row_start) or (piece.isupper() and row_end >= row_start):
    return False  # Ensure pawns are moving in the correct direction

  if board[row_end][col_end] != "" and piece.islower() == board[row_end][col_end].islower():
    return False  # Cannot move to a square occupied by a piece of the same color

  if piece[0].lower() == 'p':
    # Pawn specific rules
    if col_start == col_end and board[row_end][col_end] == "":
      # Pawn moves forward one square
      if abs(row_end - row_start) == 1:
        return True
      # Pawn initial two-square move
        elif abs(row_end - row_start) == 2 and row_start in (1, 6) and board[row_start + (1 if piece.islower() else -1)]
[col_start] == "":
        return True
    elif abs(row_end - row_start) == 1 and abs(col_end - col_start) == 1:
      # Pawn captures diagonally
      if board[row_end][col_end] != "" and piece.islower() != board[row_end][col_end].islower():
        return True
```

```
    return False

  if piece[0].lower() == 'r':
    # Rook specific rules
    return row_start == row_end or col_start == col_end and not is_obstructed(start, end, board)

  if piece[0].lower() == 'n':
    # Knight specific rules
      return (abs(row_end - row_start) == 2 and abs(col_end - col_start) == 1) or (abs(row_end - row_start) == 1 and
abs(col_end - col_start) == 2)

  if piece[0].lower() == 'b':
    # Bishop specific rules
    return abs(row_end - row_start) == abs(col_end - col_start) and not is_obstructed(start, end, board)

  if piece[0].lower() == 'q':
    # Queen specific rules
     return (row_start == row_end or col_start == col_end or abs(row_end - row_start) == abs(col_end - col_start)) and
not is_obstructed(start, end, board)

  if piece[0].lower() == 'k':
    # King specific rules
    return abs(row_end - row_start) <= 1 and abs(col_end - col_start) <= 1
```

```
  return False
```

```
def is_obstructed(start, end, board):
  row_start, col_start = start
  row_end, col_end = end

  delta_row = 1 if row_end > row_start else -1 if row_end < row_start else 0
  delta_col = 1 if col_end > col_start else -1 if col_end < col_start else 0

  current_row, current_col = row_start + delta_row, col_start + delta_col

  while (current_row, current_col) != (row_end, col_end):
    if board[current_row][current_col] != "":
      return True  # There is an obstruction
    current_row += delta_row
    current_col += delta_col

  return False
```

```
def is_in_check(board, color):
  for row in range(8):
    for col in range(8):
```

```
            piece = board[row][col]
            if piece and piece.isupper() != (color == 'w'):
                king_position = find_king(board, color)
                if is_valid_move(piece, (row, col), king_position, board):
                    return True
    return False
```

```
def find_king(board, color):
    for row in range(8):
        for col in range(8):
            if board[row][col] == f'K' if color == 'w' else 'k':
                return row, col
```

```
def is_in_checkmate(board, color):
    # Checkmate condition (stubbed)
    return False
```

```
def is_en_passant(board, start, end):
    row_start, col_start = start
    row_end, col_end = end

    # Ensure the pawn is moving two squares forward
    if board[row_start][col_start].lower() == 'p' and abs(row_end - row_start) == 2:
```

```
    # Check if there is an opponent pawn to the left or right
    if col_end > 0 and board[row_end][col_end - 1].lower() == 'p' and board[row_end][col_end - 1].isupper():
        return True
    elif col_end < 7 and board[row_end][col_end + 1].lower() == 'p' and board[row_end][col_end + 1].isupper():
        return True

  return False
```

```
def pawn_promotion(piece, end_position):
    row, col = end_position

    # Check if the pawn reached the opposite end of the board
    if piece.lower() == 'p' and (row == 0 or row == 7):
        # You might want to implement a pop-up or some UI to let the player choose the promoted piece
        promotion_piece = input(
            "Choose a piece for promotion (Q, R, N, B): ").upper()

        # Ensure the input is valid
        while promotion_piece not in ['Q', 'R', 'N', 'B']:
            promotion_piece = input(
                "Invalid choice. Choose Q, R, N, or B: ").upper()

        return promotion_piece
```

```
  return piece
```

```
running = True
while running and not is_in_checkmate(chess_board, 'w') and not is_in_checkmate(chess_board, 'b'):
  selected_piece_available_moves = []

  for i in range(8):
    for j in range(8):
      if selected_piece and selected_row is not None and selected_col is not None:
        move_valid = is_valid_move(
          selected_piece, (selected_row, selected_col), (i, j), chess_board)
        if move_valid:
          selected_piece_available_moves.append((i, j))

  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      running = False
    elif event.type == pygame.MOUSEBUTTONDOWN:
      pos = pygame.mouse.get_pos()
      col = pos[0] // SQUARE_SIZE
      row = pos[1] // SQUARE_SIZE

      if selected_piece and (row, col) in selected_piece_available_moves:
        if is_en_passant(chess_board, (selected_row, selected_col), (row, col)):
```

```
            # Handle en passant
            chess_board[row - 1 if selected_piece.islower()
                        else row + 1][col] = ""
        else:
            chess_board[row][col] = pawn_promotion(
                selected_piece, (row, col))
            chess_board[selected_row][selected_col] = ""
        selected_piece = None
        selected_row = None
        selected_col = None
    elif chess_board[row][col] != "":
        # Only set selected_piece and selected position if there's a piece in the clicked square
        selected_piece = chess_board[row][col]
        selected_row, selected_col = row, col

screen.fill((255, 255, 255))

for row in range(8):
    for col in range(8):
        color = WHITE if (row + col) % 2 == 0 else BLACK
        pygame.draw.rect(screen, color, (col * SQUARE_SIZE,
                         row * SQUARE_SIZE, SQUARE_SIZE, SQUARE_SIZE))
        piece = chess_board[row][col]
        if piece:
```

```
            color_prefix = "w" if piece.isupper() else "b"
            piece_key = color_prefix + piece.lower()
            if piece_key not in pieces:
                print(f"Piece not found: {piece_key}")
                continue

            piece_image = pieces[piece_key]
            screen.blit(
                piece_image, (col * SQUARE_SIZE, row * SQUARE_SIZE))

    if selected_piece:
        pygame.draw.rect(screen, (0, 255, 0), (selected_col * SQUARE_SIZE,
                         selected_row * SQUARE_SIZE, SQUARE_SIZE, SQUARE_SIZE), 4)

    pygame.display.flip()

# Checkmate message
if is_in_checkmate(chess_board, 'w'):
    print("Checkmate! Player B wins!")
elif is_in_checkmate(chess_board, 'b'):
    print("Checkmate! Player W wins!")

pygame.quit()
sys.exit()
```

let's go through the provided **Python Code** line by line :

Python Code

```
import pygame
import sys
import os
```

- The code starts by importing the necessary libraries : `pygame` for game development, `sys` for system - related operations, and `os` for interacting with the operating system .

Python Code

```
pygame . init ()
```

- Initializes the pygame library . This must be called before using any pygame functions .

Python Code

```
WIDTH, HEIGHT = 600, 600
BOARD_SIZE = 8
SQUARE_SIZE = WIDTH // BOARD_SIZE
WHITE = ( 255, 255, 255 )
BLACK = ( 0, 0, 0 )
```

- Sets up some constants for the dimensions of the game window, the size of the chessboard, the size of each square on the chessboard, and color constants .

Python Code

```
chess_board = [
    ["r", "n", "b", "q", "k", "b", "n", "r"],
    ["p", "p", "p", "p","p", "p", "p", "p"],
    ["", "", "", "", "", "", "", ""],
    ["", "", "", "", "", "", "", ""],
    ["", "", "", "", "", "", "", ""],
    ["", "", "", "", "", "", "", ""],
    ["P", "P", "P", "P", "P", "P", "P", "P"],
    ["R", "N", "B", "Q", "K", "B", "N", "R"],
]
```

- Initializes the chessboard as a list of lists representing the initial configuration of chess pieces .

Python Code

```
screen = pygame . display . set_mode (( WIDTH, HEIGHT ))
pygame . display . set_caption (" Chess Game ")
```

- Creates the game window with the specified dimensions and sets the window caption .

Python Code

```
pieces = {}
```

```
for color in [ 'w', 'b' ]:
    for piece in [ 'r', 'n', 'b', 'q', 'k', 'p' ]:
        img_path = os . path . join (" images " , f " {color}{piece . lower () } . png ")
        pieces [ color + piece ] = pygame . transform . scale (
                pygame . image . load ( img_path ) , ( SQUARE_SIZE, SQUARE_SIZE ))
```

- Loads chess piece images from files located in the " images " directory and scales them to match the size of a chessboard square . The images are stored in the pieces dictionary with keys like " wr " for a white rook .

Python Code

```
selected_piece = None
selected_row = None
selected_col = None
```

- Initializes variables to keep track of the currently selected chess piece and its position on the board .

The code then **def**ines several functions : is_valid_move , is_obstructed , is_in_check , find_king , is_in_checkmate , is_en_passant , and pawn_promotion . These functions are responsible for checking various conditions related to chess moves and game state .

The code then enters a game loop using a while statement, where it continuously updates the game state and checks for user input and events .

The game loop includes logic for handling mouse clicks, updating the display, and checking for checkmate conditions. The loop continues until the game is either closed or a checkmate is detected.

Finally, after exiting the game loop, the code prints a message indicating the winner (if any), closes the pygame window, and exits the program.

How To Play Chess Game

To play a chess game using the provided code, follow these general steps:

1. **Run the Code:**
    - Make sure you have Python installed on your system.
    - Save the code in a file with a `.py` extension (e.g., `chess_game.py`).
    - Open a terminal or command prompt, navigate to the directory containing the file, and run the script using `python chess_game.py`.
2. **Chessboard Display:**
    - The code will open a window displaying the chessboard and pieces.
3. **Selecting and Moving Pieces:**
    - Click on a piece to select it (highlighted by a green border).

- Click on a valid square to move the selected piece .
- If the move is valid, the piece will be moved to the new position .

4. **Pawn Promotion :**

- If a pawn reaches the opposite end of the board, you will be prompted to choose a piece for promotion (Queen, Rook, Knight, Bishop).
- Enter the corresponding letter (Q, R, N, B) to promote the pawn .

5. **Check and Checkmate :**

- The game checks for check and checkmate conditions after each move .
- If a king is in check, the board will display a message indicating the check .
- If a checkmate occurs, the game will end, and the winner will be announced .

6. **Quit the Game :**

- Close the game window to exit the program .

7. **Observing Rules :**

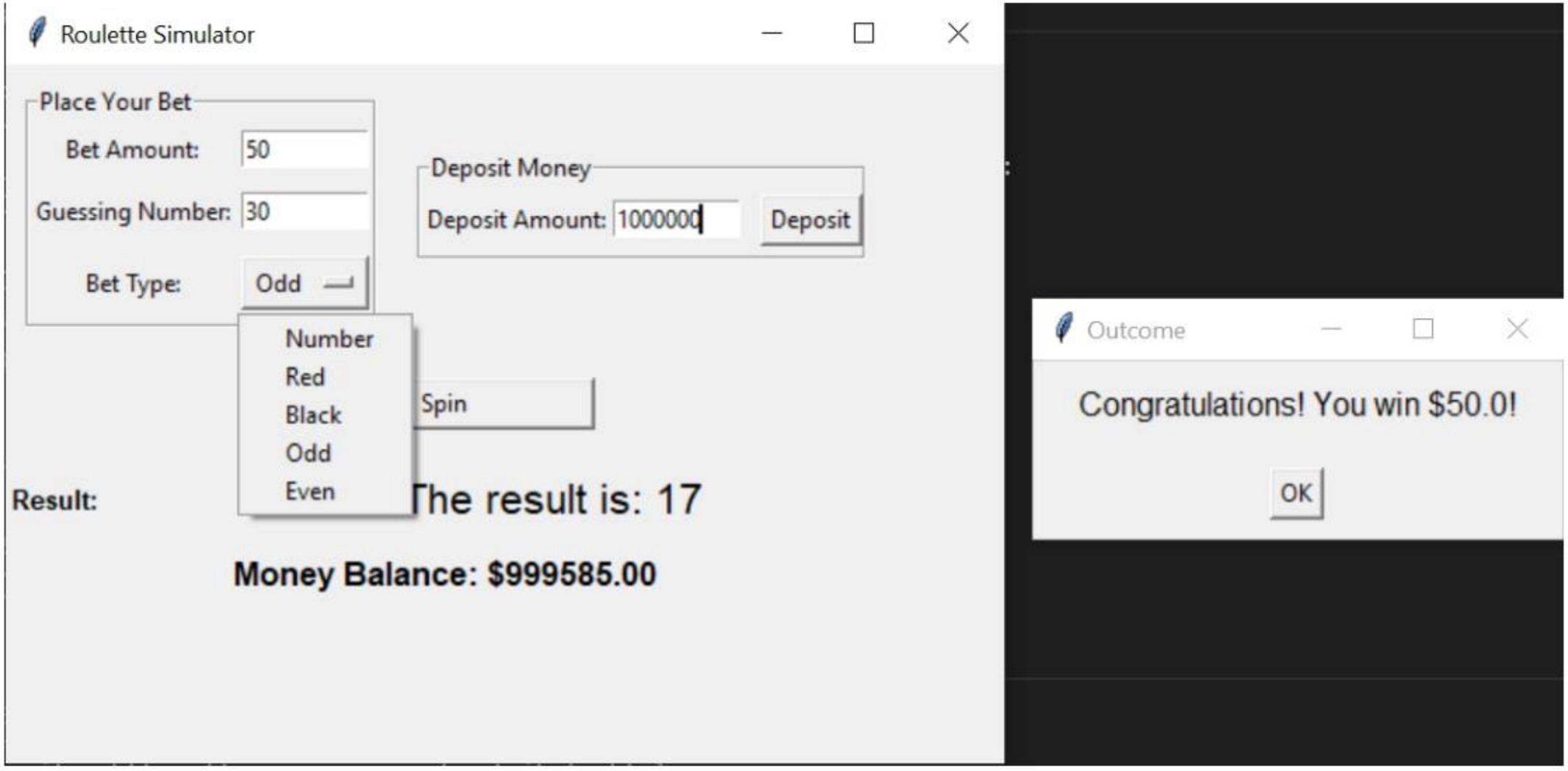
- The code enforces standard chess rules, including valid moves for each piece, castling, en passant, and pawn promotion .

8. **Customizing the Game :**

- You can modify the initial chessboard configuration or create your own custom configurations.
- Images of chess pieces are expected to be in the "images" directory. Make sure you have the necessary images for each piece (e.g., wr.png for a white rook).

Keep in mind that this code provides a basic interface for playing chess, and it's meant for educational purposes. It doesn't include features like an AI opponent, saving games, or advanced graphical elements. If you're looking for a more user-friendly and feature-rich chess-playing experience, consider using dedicated chess software or online platforms.

19. Roulette Simulator Game




```
import tkinter as tk
from tkinter import messagebox
import random

```

```
class RouletteSimulator:
```

```
def __init__(self, master):
    self.master = master
    self.master.title("Roulette Simulator")

    # Center the window on the screen
    window_width = 500
    window_height = 350
    screen_width = self.master.winfo_screenwidth()
    screen_height = self.master.winfo_screenheight()
    x_position = (screen_width - window_width) // 2
    y_position = (screen_height - window_height) // 2
    self.master.geometry(
        f"{window_width}x{window_height}+{x_position}+{y_position}")

    self.money_balance = 1000  # Starting money balance

    # Bet Controls
    self.bet_label_frame = tk.LabelFrame(master, text="Place Your Bet")
    self.bet_label_frame.grid(
        row=0, column=0, padx=10, pady=10, sticky="w")

    self.bet_amount_label = tk.Label(
        self.bet_label_frame, text="Bet Amount:")
    self.bet_amount_label.grid(row=0, column=0, pady=5)
```

```
self.bet_amount_var = tk.StringVar()
self.bet_amount_entry = tk.Entry(
  self.bet_label_frame, textvariable=self.bet_amount_var, width=10)
self.bet_amount_entry.grid(row=0, column=1, pady=5)

self.bet_number_label = tk.Label(
  self.bet_label_frame, text="Guessing Number:")
self.bet_number_label.grid(row=1, column=0, pady=5)

self.bet_number_var = tk.StringVar()
self.bet_number_entry = tk.Entry(
  self.bet_label_frame, textvariable=self.bet_number_var, width=10)
self.bet_number_entry.grid(row=1, column=1, pady=5)

self.bet_type_label = tk.Label(self.bet_label_frame, text="Bet Type:")
self.bet_type_label.grid(row=2, column=0, pady=5)

self.bet_type_var = tk.StringVar()
self.bet_type_var.set("Number")
bet_types = ["Number", "Red", "Black", "Odd", "Even"]
self.bet_type_menu = tk.OptionMenu(
  self.bet_label_frame, self.bet_type_var, *bet_types)
self.bet_type_menu.grid(row=2, column=1, pady=5)
```

```
# Deposit Controls
self.deposit_label_frame = tk.LabelFrame(master, text="Deposit Money")
self.deposit_label_frame.grid(
  row=0, column=1, padx=10, pady=10, sticky="w")

self.deposit_label = tk.Label(
  self.deposit_label_frame, text="Deposit Amount:")
self.deposit_label.grid(row=0, column=0, pady=5)

self.deposit_var = tk.StringVar()
self.deposit_entry = tk.Entry(
  self.deposit_label_frame, textvariable=self.deposit_var, width=10)
self.deposit_entry.grid(row=0, column=1, pady=5)

self.deposit_button = tk.Button(
  self.deposit_label_frame, text="Deposit", command=self.deposit_money)
self.deposit_button.grid(row=0, column=2, pady=5, padx=(10, 0))

# Spin Button
self.spin_button = tk.Button(
  master, text="Spin", command=self.start_spin, width=20, state='disabled')
self.spin_button.grid(row=1, column=0, columnspan=2, pady=15)
```

```
# Result and Balance Labels
self.result_label = tk.Label(
  master, text="Result:", font=("Helvetica", 10, "bold"))
self.result_label.grid(row=2, column=0, pady=(5, 0), sticky="w")

self.result_var = tk.StringVar()
self.result_value_label = tk.Label(
  master, textvariable=self.result_var, font=("Helvetica", 16))
self.result_value_label.grid(row=2, column=1, pady=(5, 0), sticky="w")

self.balance_label = tk.Label(
  master, text=f"Money Balance: ${self.money_balance:.2f}", font=("Helvetica", 12, "bold"))
self.balance_label.grid(row=3, column=0, columnspan=2, pady=(10, 0))

self.is_spinning = False
self.spin_interval = 100  # milliseconds
self.spin_count = 0
self.stop_spin_count = 10

# Trace changes in bet amount entry to enable/disable Spin Button
self.bet_amount_var.trace('w', self.check_bet_amount)

# Trace changes in bet number entry to enable/disable Spin Button
self.bet_number_var.trace('w', self.check_bet_amount)
```

```
def deposit_money(self):
  try:
    deposit_amount = float(self.deposit_var.get())
    if deposit_amount <= 0:
      messagebox.showerror(
        "Error", "Deposit amount must be greater than zero.")
      return
  except ValueError:
    messagebox.showerror(
      "Error", "Invalid deposit amount. Please enter a valid number.")
    return

  self.money_balance += deposit_amount
  self.update_balance_label()

def start_spin(self):
  if not self.is_spinning:
    self.spin_count = 0
    self.is_spinning = True
    self.spin()

def spin(self):
  if self.is_spinning:
```

```
    # 0 to 36 for numbers, 37 for '00'
    result = random.choice(range(37))
    self.result_var.set(result)
    self.spin_count += 1

    if self.spin_count < self.stop_spin_count:
      self.master.after(self.spin_interval, self.spin)
    else:
      self.is_spinning = False
      self.process_spin_result(result)

def process_spin_result(self, result):
  try:
    bet_amount = float(self.bet_amount_var.get())
    bet_number = self.bet_number_var.get()
    if bet_amount <= 0 or bet_amount > self.money_balance or not bet_number:
      messagebox.showerror(
        "Error", "Invalid bet amount or guessing number. Please enter valid values.")
      return
  except ValueError:
    messagebox.showerror(
      "Error", "Invalid bet amount. Please enter a valid number.")
    return
```

```
    bet_type = self.bet_type_var.get()
    self.result_var.set(f"The result is: {result}")

    win_amount = self.check_win(bet_type, result, bet_amount, bet_number)
    self.money_balance += win_amount - bet_amount
    self.update_balance_label()

    if win_amount > 0:
        self.show_outcome_message(
            f"Congratulations! You win ${win_amount - bet_amount}!")
    else:
        self.show_outcome_message("You lose!")

def show_outcome_message(self, message):
    outcome_window = tk.Toplevel(self.master)
    outcome_window.title("Outcome")
    outcome_window.geometry(
        "+%d+%d" % (self.master.winfo_x() + 500, self.master.winfo_y() + 130))

    outcome_label = tk.Label(
        outcome_window, text=message, font=("Helvetica", 12))
    outcome_label.pack(padx=20, pady=10)

    ok_button = tk.Button(outcome_window, text="OK",
```

```
                command=outcome_window.destroy)
    ok_button.pack(pady=10)

def check_win(self, bet_type, result, bet_amount, bet_number):
    if bet_type == "Number":
        try:
            selected_number = int(bet_number)
        except ValueError:
            messagebox.showerror(
                "Error", "Invalid bet number. Please enter a valid number.")
            return 0
        if selected_number == result:
            return bet_amount * 36  # Winning on a specific number
        else:
            return 0
    elif bet_type == "Red" and result in [1, 3, 5, 7, 9, 12, 14, 16, 18, 19, 21, 23, 25, 27, 30, 32, 34, 36]:
        return bet_amount * 2  # Winning on red
    elif bet_type == "Black" and result in [2, 4, 6, 8, 10, 11, 13, 15, 17, 20, 22, 24, 26, 28, 29, 31, 33, 35]:
        return bet_amount * 2  # Winning on black
    elif bet_type == "Odd" and result % 2 != 0:
        return bet_amount * 2  # Winning on odd
    elif bet_type == "Even" and result % 2 == 0:
        return bet_amount * 2  # Winning on even
    else:
```

```
        return 0

    def update_balance_label(self):
        self.balance_label.config(
            text=f"Money Balance: ${self.money_balance:.2f}")

    def check_bet_amount(self, *args):
        try:
            bet_amount = float(self.bet_amount_var.get())
            bet_number = self.bet_number_var.get()
            if bet_amount <= 0 or bet_amount > self.money_balance or not bet_number:
                self.spin_button['state'] = 'disabled'
            else:
                self.spin_button['state'] = 'normal'
        except ValueError:
            self.spin_button['state'] = 'disabled'

if __name__ == "__main__":
    root = tk.Tk()
    app = RouletteSimulator(root)
    root.mainloop()
```

let's go through the code line by line :

1. import tkinter as tk : This imports the Tkinter module, which provides a toolkit for creating graphical user interfaces .
2. from tkinter import messagebox : This imports the messagebox module from Tkinter, which is used to display pop - up message boxes .
3. import random : This imports the random module, which is used for generating random numbers .
4. class RouletteSimulator :: **Def**ines a class named RouletteSimulator to encapsulate the functionality of the roulette simulator .
5. **def** __init__ (self, master):: Initializes the class . The master parameter is a Tkinter root window or another Tkinter widget that serves as the main window .
6. self . master = master : Stores the reference to the Tkinter root window or main widget .
7. self . master . title (" Roulette Simulator "): Sets the title of the main window .
8. Window Geometry Configuration :
    - window_width = 500 : Sets the initial width of the window .
    - window_height = 350 : Sets the initial height of the window .

- Calculates the position to center the window on the screen .
- self . master . geometry (...) : Sets the window size and position .

9. self . money_balance = 1000 : Initializes the starting money balance .
10. Bet Controls Section :
    - self . bet_label_frame : Creates a labeled frame for bet - related controls .
    - self . bet_amount_label , self . bet_number_label , self . bet_type_label : Labels for bet amount, guessing number, and bet type .
    - self . bet_amount_var , self . bet_number_var , self . bet_type_var : StringVars to store user inputs .
    - self . bet_amount_entry , self . bet_number_entry : Entry widgets for entering bet amount and guessing number .
    - self . bet_type_menu : OptionMenu for selecting the bet type .
11. Deposit Controls Section :
    - self . deposit_label_frame : Creates a labeled frame for deposit - related controls .
    - self . deposit_label : Label for deposit amount .
    - self . deposit_var : StringVar to store deposit amount .

- self . deposit_entry : Entry widget for entering deposit amount .
- self . deposit_button : Button to trigger the deposit process .

12. Spin Button :

- self . spin_button : Button for spinning the roulette wheel . Initially disabled .

13. Result and Balance Labels :

- self . result_label , self . result_value_label : Labels for displaying the result .
- self . balance_label : Label for displaying the money balance .

14. Spin - related variables :

- self . is_spinning , self . spin_interval , self . spin_count , self . stop_spin_count : Variables for controlling the spinning process .

15. Trace changes in bet amount and bet number to enable / disable the Spin Button .

16. **def** deposit_money (self):: Method for handling the deposit of money .

17. **def** start_spin (self):: Method to initiate the spinning process .

18. **def** spin (self):: Method for simulating the spinning of the roulette wheel .

19. **def** process_spin_result (self, result):: Method for processing the spin result and updating the balance .

20. **def** show_outcome_message (self, message):: Method for displaying an outcome message in a separate window .

21. **def** check_win (self, bet_type, result, bet_amount, bet_number):: Method to check if the user won and calculate the winning amount .

22. **def** update_balance_label (self):: Method to update the money balance label .

23. **def** check_bet_amount (self, * args):: Method to check the validity of the bet amount and enable / disable the Spin Button .

24. Main block :

- Creates a Tkinter root window .
- Instantiates the RouletteSimulator class .
- Enters the Tkinter main event loop .

How To Play Roulette Simulator Game

To play the Roulette Simulator game :

1. **Launch the Game :**

- Run the Python script to launch the game .
- The game window will appear with various controls for placing bets, depositing money, and spinning the roulette wheel .

2. **Place Your Bet :**

- In the " Place Your Bet " section, enter the amount you want to bet in the " Bet Amount " entry .
- Choose the type of bet you want to place using the " Bet Type " dropdown menu . Options include :
  - Number : Bet on a specific number (0 to 36).
  - Red : Bet on red numbers .
  - Black : Bet on black numbers .
  - Odd : Bet on odd numbers .
  - Even : Bet on even numbers .
- Depending on the selected bet type, additional input may be required (e . g . , guessing a specific number).

3. **Deposit Money :**

- In the " Deposit Money " section, enter the amount you want to deposit in the " Deposit Amount " entry .
- Click the " Deposit " button to add money to your balance .

4. **Spin the Wheel :**

- Once you've placed your bet and deposited money, the " Spin " button becomes enabled .
- Click the " Spin " button to start the roulette wheel .

5. **View the Result :**

- The roulette wheel will spin, and after a few moments, the result will be displayed in the " Result " section .
- The outcome will show whether you won or lost and the specific result .

6. **Outcome Message :**

- A pop - up window will appear with a message indicating whether you won or lost .
- If you won, it will also display the amount you won .

7. **Repeat or Adjust :**

- You can repeat the process by placing new bets, changing bet types, or depositing more money .
- Adjust your bets and strategy based on the outcomes and your remaining balance .

8. **Quit the Game :**

- Close the main window to exit the game .

Remember that this is a simplified roulette simulator, and the goal is to enjoy the experience of betting and spinning the wheel without real money involved . Have fun exploring different bet types and strategies !

20. Mancala Game


```
import tkinter as tk
from tkinter import ttk, messagebox
from ttkthemes import ThemedStyle
```

```
class MancalaGame:
  def __init__(self, master):
    self.master = master
    self.master.title("Mancala Game")

    # Mancala board representation
    # 8 pits for each player + 2 mancalas for each player
```

```
self.board = [4] * 16

# Player 1's side
self.p1_pits = [tk.Button(master, text=str(self.board[i]), command=lambda i=i: self.move(i), font=('Arial', 10))
        for i in range(8)]

# Player 2's side
self.p2_pits = [tk.Button(master, text=str(self.board[i]), command=lambda i=i: self.move(i), font=('Arial', 10))
        for i in range(8, 16)]

# Mancalas
self.p1_mancala = tk.Label(
  master, text="0", font=('Arial', 12, 'bold'))
self.p2_mancala = tk.Label(
  master, text="0", font=('Arial', 12, 'bold'))

# Score labels
self.score_label_p1 = tk.Label(
  master, text="Player 1 Score:", font=('Arial', 10, 'italic'))
self.score_label_p2 = tk.Label(
  master, text="Player 2 Score:", font=('Arial', 10, 'italic'))

# Reset button
self.reset_button = tk.Button(
```

```
        master, text="Reset Game", command=self.reset_game, font=('Arial', 10, 'bold'))

    # Create the GUI layout
    self.create_layout()

    # Track the current player
    self.current_player = 1
    self.extra_turn = False

def create_layout(self):
    style = ThemedStyle(self.master)
    style.set_theme("plastik")  # You can choose other available themes

    # Player 2's pits and mancala
    for i in range(8):
        self.p2_pits[i].grid(row=1, column=i, padx=5, pady=5)
    self.p2_mancala.grid(row=1, column=9, padx=10, pady=5)

    # Player 1's pits and mancala
    for i in range(8):
        self.p1_pits[i].grid(row=2, column=7-i, padx=5, pady=5)
    self.p1_mancala.grid(row=2, column=0, padx=10, pady=5)

    # Score labels
```

```
    self.score_label_p1.grid(
        row=3, column=0, padx=10, pady=5, columnspan=4)
    self.score_label_p2.grid(
        row=3, column=5, padx=10, pady=5, columnspan=4)

    # Reset button
    self.reset_button.grid(row=4, column=0, columnspan=10, pady=10)

def move(self, pit_index):
    if self.current_player == 1 and pit_index < 8:
        self.make_move(pit_index)
    elif self.current_player == 2 and 8 <= pit_index <= 15:
        self.make_move(pit_index)
    else:
        messagebox.showinfo("Invalid Move", "It's not your turn!")

def make_move(self, pit_index):
    stones = self.board[pit_index]
    self.board[pit_index] = 0

    while stones > 0:
        pit_index = (pit_index + 1) % 16
        if self.current_player == 1 and pit_index == 15:
            continue  # skip opponent's mancala
```

```
        elif self.current_player == 2 and pit_index == 8:
            continue  # skip opponent's mancala

        self.board[pit_index] += 1
        stones -= 1

    self.update_gui()
    self.check_extra_turn(pit_index)
    self.check_end_game()

    if not self.extra_turn:
        # Switch player only if there is no extra turn
        # Switch between player 1 and player 2
        self.current_player = 3 - self.current_player

def update_gui(self):
    # Update Player 1's side
    for i in range(8):
        self.p1_pits[i]["text"] = str(self.board[i])

    self.p1_mancala["text"] = str(self.board[8])

    # Update Player 2's side
    for i in range(8, 16):
```

```
            self.p2_pits[i-8]["text"] = str(self.board[i])

        self.p2_mancala["text"] = str(self.board[15])

        # Update scores
        self.score_label_p1["text"] = f"Player 1 Score: {sum(self.board[:8])}"
        self.score_label_p2["text"] = f"Player 2 Score: {sum(self.board[8:16])}"

    def check_extra_turn(self, last_pit_index):
        if self.current_player == 1 and 0 <= last_pit_index < 8 and self.board[last_pit_index] == 1:
            self.extra_turn = True
        elif self.current_player == 2 and 8 <= last_pit_index < 15 and self.board[last_pit_index] == 1:
            self.extra_turn = True
        else:
            self.extra_turn = False

    def check_end_game(self):
        if all(pit == 0 for pit in self.board[:8]) or all(pit == 0 for pit in self.board[8:16]):
            self.end_game()

    def end_game(self):
        p1_score = sum(self.board[:8])
        p2_score = sum(self.board[8:16])
```

```
        if p1_score > p2_score:
            winner = "Player 1"
        elif p1_score < p2_score:
            winner = "Player 2"
        else:
            winner = "It's a tie!"

        messagebox.showinfo("Game Over", f"The game is over!\n{winner} wins!")

    def reset_game(self):
        self.board = [4] * 16
        self.current_player = 1
        self.extra_turn = False
        self.update_gui()
```

```
if __name__ == "__main__":
    root = tk.Tk()
    mancala_game = MancalaGame(root)
    root.mainloop()
```

Let's go through the code line by line to understand its functionality :

Python Code

```
import tkinter as tk
from tkinter import ttk, messagebox
from ttkthemes import ThemedStyle
```

Here, the code imports the necessary modules from the tkinter library for creating a graphical user interface (GUI) , including themed styling .

Python Code

```
class MancalaGame :
    def __init__ ( self, master ):
        self . master = master
        self . master . title (" Mancala Game ")
```

A class MancalaGame is **def**ined to encapsulate the Mancala game . The __init__ method initializes the game, setting up the main window (master) and setting its title .

Python Code

```
        # Mancala board representation
        # 8 pits for each player + 2 mancalas for each player
        self . board = [ 4 ]* 16
```

The Mancala board is represented as a list (self . board) containing 16 pits - 8 pits for each player and 2 mancalas for each player . The initial configuration is set with 4 stones in each pit .

Python Code

```
        # Player 1's side
    self . p1_pits          =          [ tk . Button ( master,          text = str ( self . board [ i ]) ,
command = lambdai = i :  self . move ( i ) , font =( 'Arial', 10 ))
                            for i in range ( 8 )]
```

A list of 8 buttons (self . p1_pits) is created for Player 1's side, representing the pits . The buttons display the number of stones in each pit, and the command parameter is set to the self . move method with the current pit index .

Python Code

```
        # Player 2's side
    self . p2_pits          =          [ tk . Button ( master,          text = str ( self . board [ i ]) ,
command = lambdai = i :  self . move ( i ) , font =( 'Arial', 10 ))
                            for i in range ( 8, 16 )]
```

Similarly, a list of 8 buttons (self . p2_pits) is created for Player 2's side .

Python Code

```
        # Mancalas
    self . p1_mancala  =  tk . Label (
            master, text =" 0 " , font =( 'Arial', 12, 'bold' ))
    self . p2_mancala  =  tk . Label (
            master, text =" 0 " , font =( 'Arial', 12, 'bold' ))
```

Labels are created to represent the mancalas for both players . They initially display the number 0 .

Python Code

```
        # Score labels
    self . score_label_p1  =  tk . Label (
            master, text =" Player 1 Score :" , font =( 'Arial', 10, 'italic' ))
    self . score_label_p2  =  tk . Label (
            master, text =" Player 2 Score :" , font =( 'Arial', 10, 'italic' ))
```

Labels are created to display the scores for Player 1 and Player 2 .

Python Code

```
        # Reset button
    self . reset_button  =  tk . Button (
```

master, text =" Reset Game " , command = self . reset_game, font =('Arial', 10, 'bold'))

A button (self . reset_button) is created to reset the game, and its command is set to the self . reset_game method .

Python Code

```
    # Create the GUI layout
self . create_layout ()
```

The create_layout method is called to organize and place the widgets on the GUI .

Python Code

```
    # Track the current player
self . current_player = 1
self . extra_turn = False
```

Variables self . current_player and self . extra_turn are initialized to track the current player and whether an extra turn is granted .

Python Code

```
def create_layout ( self ):
    style = ThemedStyle ( self . master )
    style . set_theme (" plastik ") # You can choose other available themes
```

The create_layout method sets the theme for the GUI using the ThemedStyle from the ttkthemes library .

Python Code

```
    # Player 2's pits and mancala
for i in range ( 8 ):
        self . p2_pits [ i ]. grid ( row = 1, column = i, padx = 5, pady = 5 )
self . p2_mancala . grid ( row = 1, column = 9, padx = 10, pady = 5 )
```

The buttons representing Player 2's pits and the mancala are placed on the GUI grid .

Python Code

```
    # Player 1's pits and mancala
for i in range ( 8 ):
        self . p1_pits [ i ]. grid ( row = 2, column = 7 - i, padx = 5, pady = 5 )
self . p1_mancala . grid ( row = 2, column = 0, padx = 10, pady = 5 )
```

Similarly, the buttons representing Player 1's pits and the mancala are placed on the GUI grid .

Python Code

```
    # Score labels
self . score_label_p1 . grid (
```

```
            row = 3, column = 0, padx = 10, pady = 5, columnspan = 4 )
        self . score_label_p2 . grid (
            row = 3, column = 5, padx = 10, pady = 5, columnspan = 4 )
```

The labels displaying the scores for both players are placed on the GUI grid .

Python Code

```
        # Reset button
    self . reset_button . grid ( row = 4, column = 0, columnspan = 10, pady = 10 )
```

The reset button is placed on the GUI grid .

Python Code

```
def move ( self, pit_index ):
    if self . current_player == 1 and pit_index < 8 :
            self . make_move ( pit_index )
    elif self . current_player == 2 and 8< = pit_index < = 15 :
            self . make_move ( pit_index )
    else :
            messagebox . showinfo (" Invalid Move " , " It's not your turn !")
```

The move method is called when a pit button is clicked . It checks whether the move is valid for the current player and calls the make_move method .

Python Code

```
def make_move ( self, pit_index ):
    stones = self . board [ pit_index ]
    self . board [ pit_index ] = 0

    while stones > 0 :
            pit_index = ( pit_index + 1 ) % 16
            if self . current_player == 1 and pit_index == 15 :
                continue # skip opponent's mancala
```

```
            elif self . current_player == 2 and pit_index == 8 :
                continue # skip opponent's mancala

            self . board [ pit_index ] += 1
            stones -= 1

        self . update_gui ()
        self . check_extra_turn ( pit_index )
        self . check_end_game ()

        if not self . extra_turn :
            # Switch player only if there is no extra turn
                # Switch between player 1 and player 2
            self . current_player = 3 - self . current_player
```

The make_move method handles the logic of distributing stones in the pits after a move is made . It updates the GUI, checks for an extra turn, and checks if the game has ended .

Python Code

```
    def update_gui ( self ):
        # Update Player 1's side
        for i in range ( 8 ):
```

```
            self . p1_pits [ i ][" text "] =  str ( self . board [ i ])

    self . p1_mancala [" text "] =  str ( self . board [ 8 ])

    # Update Player 2's side
    for i in range ( 8, 16 ):
            self . p2_pits [ i - 8 ][" text "] =  str ( self . board [ i ])

    self . p2_mancala [" text "] =  str ( self . board [ 15 ])

    # Update scores
    self . score_label_p1 [" text "] =  f " Player 1 Score :  {sum ( self . board [: 8 ]) } "
    self . score_label_p2 [" text "] =  f " Player 2 Score :  {sum ( self . board [ 8 : 16 ]) } "
```

The update_gui method updates the text on the buttons and labels to reflect the current state of the game .

Python Code

```
  def check_extra_turn ( self, last_pit_index ):
    if self . current_player == 1 and 0< = last_pit_index < 8 and
self . board [ last_pit_index ] == 1 :
            self . extra_turn = True
```

```
    elif self.current_player == 2 and 8<= last_pit_index < 15 and
self.board[last_pit_index] == 1:
            self.extra_turn = True
    else:
            self.extra_turn = False
```

The check_extra_turn method determines whether an extra turn is granted based on the last pit index where a stone was placed .

Python Code

```
  def check_end_game(self):
    if all(pit == 0 for pit in self.board[:8]) or all(pit == 0 for pit in
self.board[8:16]):
            self.end_game()
```

The check_end_game method checks if the game has ended by examining whether all pits on one side are empty .

Python Code

```
  def end_game(self):
    p1_score = sum(self.board[:8])
    p2_score = sum(self.board[8:16])
```

```
    if p1_score > p2_score :
            winner  = " Player 1 "
    elif p1_score < p2_score :
            winner  = " Player 2 "
    else :
            winner  = " It's a tie !"

    messagebox . showinfo (" Game Over " , f " The game is over ! \n{winner} wins !")
```

The end_game method displays a message box announcing the end of the game and the winner or a tie .

Python Code

```
  def reset_game ( self ):
    self . board  = [ 4 ]* 16
    self . current_player  = 1
    self . extra_turn  = False
    self . update_gui ()
```

The reset_game method resets the game state by setting the board to its initial configuration, resetting the current player and extra turn variables, and updating the GUI .

Python Code

```
if __name__ == " __main__ ":
    root = tk . Tk ()
    mancala_game = MancalaGame ( root )
    root . mainloop ()
```

The script creates a Tkinter root window, initializes an instance of the MancalaGame class, and starts the main event loop using root . mainloop (). This loop keeps the GUI responsive to user interactions .

How To Play Mancala Game

Mancala is a two - player strategy board game that involves capturing stones or seeds in pits on the game board . The game typically starts with a certain number of stones or seeds in each pit . Here's a basic guide on how to play Mancala :

Objective : The goal of Mancala is to capture more stones or seeds than your opponent .

Setup :

1. The Mancala board consists of two rows of six pits each, for a total of 12 pits, and each player controls one row of six pits .
2. At the ends of the board, each player has a larger pit called the " Mancala ."

3. Place an equal number of stones or seeds in each of the 12 smaller pits . A common starting configuration is four stones in each pit .

Starting the Game :

1. Players sit opposite each other, facing the board, with their Mancalas on their right - hand side .
2. Decide who goes first . Players may use a coin toss, rock - paper - scissors, or any other method to determine the starting player .

Gameplay :

1. On a player's turn, they select one of the pits from their row that contains stones or seeds .
2. The player then picks up all the stones or seeds from the chosen pit and distributes them, one by one, in a counterclockwise direction into the succeeding pits, including their own Mancala but skipping the opponent's Mancala .
3. If the last stone or seed is dropped into the player's Mancala, they get another turn . If the last stone lands in an empty pit on their side, the player captures that stone and any stones in the opponent's pit directly opposite . These captured stones are placed in the player's Mancala .
4. The game continues with players taking turns until one side of the board is empty .

Ending the Game : The game ends when one player no longer has stones or seeds in their pits . The remaining stones on the opposite side of the board are captured by the other player . The player with the most stones or seeds in their Mancala is declared the winner .

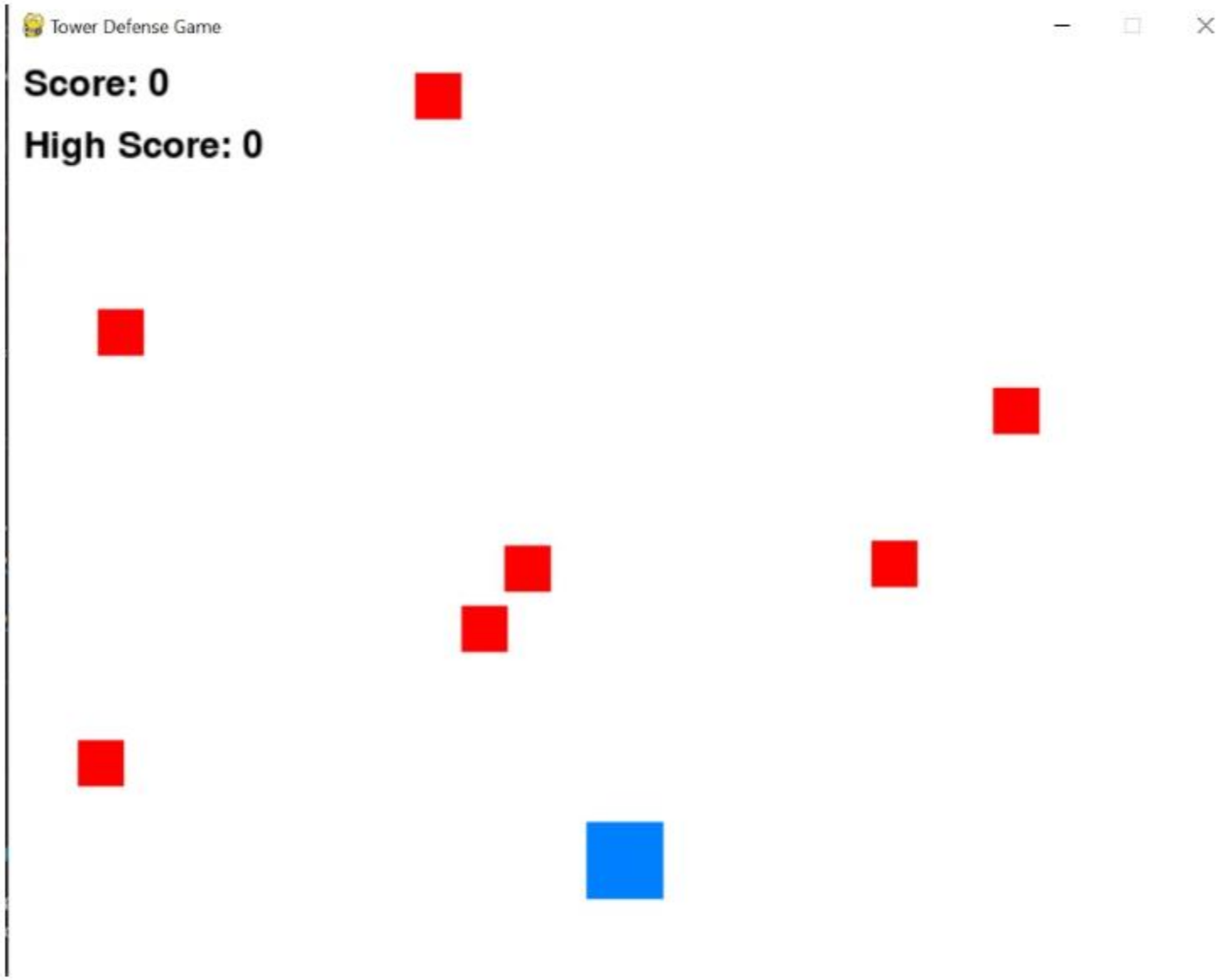
Winning : The player with the most stones or seeds in their Mancala at the end of the game is the winner . If the Mancalas have an equal number of stones or seeds, the game is a tie .

Tips :

- Pay attention to the number of stones or seeds in each pit to plan strategic moves .
- Think about capturing your opponent's stones by landing the last stone in an empty pit on your side .

Mancala is a game that combines skill and strategy, making each move crucial to the outcome . Enjoy playing and have fun !

21. Tower Defense game




```
import pygame
import sys
import random

# Initialize Pygame
```

```
pygame.init()

# Constants
WIDTH, HEIGHT = 800, 600
FPS = 60
WHITE = (255, 255, 255)
RED = (255, 0, 0)
BLUE = (0, 128, 255)
YELLOW = (255, 255, 0)

# Set up the game window
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Tower Defense Game")
clock = pygame.time.Clock()

# Player
player_size = 50
player_pos = [WIDTH // 2 - player_size // 2, HEIGHT - player_size * 2]
player_color = BLUE
player_speed = 5
can_shoot = True

# Towers
tower_size = 30
```

```
tower_color = (0, 255, 0)
towers = []

# Bullets
bullet_size = 10
bullet_color = YELLOW
bullets = []
bullet_speed = 8

# Enemies
enemy_size = 30
enemy_color = RED
enemy_speed = 3
enemies = []

# Score
score = 0
high_score = 0
font = pygame.font.Font(None, 36)

# Game state
game_over = False
restart_message = font.render("Game Over! Restart Please Press R", True, RED)
restart_message_rect = restart_message.get_rect(
```

```
  center=(WIDTH // 2, HEIGHT // 2))

# Pause state
pause = False
pause_message = font.render("Game Paused. Press P to Resume", True, BLUE)
pause_message_rect = pause_message.get_rect(center=(WIDTH // 2, HEIGHT // 2))

# Game loop
while True:
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      pygame.quit()
      sys.exit()
    elif event.type == pygame.KEYDOWN:
      if event.key == pygame.K_r and game_over:
        # Reset the game
        game_over = False
        enemies.clear()
        bullets.clear()
        score = 0
      elif event.key == pygame.K_SPACE and not game_over:
        # Shoot a bullet from the player's position
        bullet_pos = [player_pos[0] + player_size // 2, player_pos[1]]
        bullets.append(bullet_pos)
```

```
        elif event.key == pygame.K_p and not game_over:
            # Toggle pause
            pause = not pause

    keys = pygame.key.get_pressed()
    if not pause and not game_over:  # Check if the game is not paused and not game over
        if keys[pygame.K_LEFT] and player_pos[0] > 0:
            player_pos[0] -= player_speed
        if keys[pygame.K_RIGHT] and player_pos[0] < WIDTH - player_size:
            player_pos[0] += player_speed

    if not game_over and not pause:
        for enemy in enemies:
            enemy[1] += enemy_speed

        bullets = [[bullet[0], bullet[1] - bullet_speed] for bullet in bullets]
        bullets = [bullet for bullet in bullets if 0 < bullet[1] < HEIGHT]

        for bullet in bullets[:]:
            for enemy in enemies[:]:
                if (
                    enemy[0] < bullet[0] < enemy[0] + enemy_size
                    and enemy[1] < bullet[1] < enemy[1] + enemy_size
                ):
```

```
            bullets.remove(bullet)
            enemies.remove(enemy)
            score += 10
            break

    for enemy in enemies:
        if (
            player_pos[0] < enemy[0] + enemy_size
            and player_pos[0] + player_size > enemy[0]
            and player_pos[1] < enemy[1] + enemy_size
            and player_pos[1] + player_size > enemy[1]
        ):
            game_over = True
            if score > high_score:
                high_score = score
            break

    if random.randint(0, 100) < 5:
        enemy_pos = [random.randint(0, WIDTH - enemy_size), 0]
        enemies.append(enemy_pos)

    enemies = [enemy for enemy in enemies if enemy[1] < HEIGHT]

# Draw
```

```
screen.fill(WHITE)

pygame.draw.rect(screen, player_color,
          (player_pos[0], player_pos[1], player_size, player_size))

for tower in towers:
  pygame.draw.rect(screen, tower_color,
            (tower[0], tower[1], tower_size, tower_size))

for bullet in bullets:
  pygame.draw.circle(screen, bullet_color, (int(
    bullet[0]), int(bullet[1])), bullet_size)

for enemy in enemies:
  pygame.draw.rect(screen, enemy_color,
            (enemy[0], enemy[1], enemy_size, enemy_size))

score_text = font.render(f"Score: {score}", True, (0, 0, 0))
screen.blit(score_text, (10, 10))

high_score_text = font.render(f"High Score: {high_score}", True, (0, 0, 0))
screen.blit(high_score_text, (10, 50))

if game_over:
```

```
    screen.blit(restart_message, restart_message_rect)
elif pause:
    screen.blit(pause_message, pause_message_rect)

pygame.display.flip()
clock.tick(FPS)
```

Let's go through the code line by line to understand its functionality :

Python Code

```
import pygame
import sys
import random
```

- The code begins by importing the necessary modules : pygame for game development, sys for system - specific parameters and functions, and random for generating random numbers .

Python Code

```
# Initialize Pygame
pygame . init ()
```

- Pygame is initialized to set up the gaming environment .

Python Code

```
# Constants
WIDTH, HEIGHT = 800, 600
FPS = 60
WHITE = ( 255, 255, 255 )
RED = ( 255, 0, 0 )
BLUE = ( 0, 128, 255 )
YELLOW = ( 255, 255, 0 )
```

- Constants are **def**ined, including the game window dimensions (WIDTH and HEIGHT) , frames per second (FPS) , and various color constants in RGB format .

Python Code

```
# Set up the game window
screen = pygame . display . set_mode (( WIDTH, HEIGHT ))
pygame . display . set_caption (" Tower Defense Game ")
clock = pygame . time . Clock ()
```

- The game window is initialized using Pygame . A caption is set, and a clock object is created to control the frame rate .

Python Code

```
# Player
player_size = 50
player_pos = [ WIDTH // 2 - player_size // 2, HEIGHT - player_size * 2 ]
player_color = BLUE
player_speed = 5
can_shoot = True
```

- Player - related variables are **def**ined, such as size, initial position, color, speed, and a flag (can_shoot) indicating whether the player can shoot .

Python Code

```
# Towers
tower_size = 30
tower_color = ( 0, 255, 0 )
towers = []
```

- Tower - related variables are **def**ined, including size, color, and an empty list (towers) to store tower positions .

Python Code

```
# Bullets
```

```
bullet_size = 10
bullet_color = YELLOW
bullets = []
bullet_speed = 8
```

- Bullet - related variables are **def**ined, including size, color, an empty list (bullets) to store bullet positions, and the speed of bullets .

Python Code

```
# Enemies
enemy_size = 30
enemy_color = RED
enemy_speed = 3
enemies = []
```

- Enemy - related variables are **def**ined, including size, color, speed, and an empty list (enemies) to store enemy positions .

Python Code

```
# Score
score = 0
high_score = 0
```

```
font = pygame . font . Font ( None, 36 )
```

- Score - related variables are **def**ined, including the current score, high score, and a font object for rendering text .

Python Code

```
# Game state
game_over = False
restart_message = font . render (" Game Over ! Restart Please Press R " , True, RED )
restart_message_rect = restart_message . get_rect ( center =( WIDTH // 2, HEIGHT // 2 ))
```

- The game state is initialized, starting with game_over set to False . A message for restarting after game over is created, and its position is set at the center of the screen .

Python Code

```
# Pause state
pause = False
pause_message = font . render (" Game Paused . Press P to Resume " , True, BLUE )
pause_message_rect = pause_message . get_rect ( center =( WIDTH // 2, HEIGHT // 2 ))
```

- The pause state is initialized, starting with pause set to False . A message for pausing is created, and its position is set at the center of the screen .

Python Code

```
# Game loop
while True :
```

- The main game loop begins .

Python Code

```
    for event in pygame . event . get ():
        if event . type ==  pygame . QUIT :
                pygame . quit ()
                sys . exit ()
        elif event . type ==  pygame . KEYDOWN :
                if event . key  ==  pygame . K_r and game_over :
                    # Reset the game
                    game_over  =  False
                    enemies . clear ()
                    bullets . clear ()
                    score  =  0
                elif event . key  ==  pygame . K_SPACE and not game_over :
                    # Shoot a bullet from the player's position
```

```
            bullet_pos = [player_pos[0] + player_size // 2, player_pos[1]]
            bullets.append(bullet_pos)
        elif event.key == pygame.K_p and not game_over:
            # Toggle pause
            pause = not pause
```

- The event loop checks for user input, including quitting the game, restarting the game after it's over, shooting bullets with the space key, and toggling pause with the 'p' key.

Python Code

```
keys = pygame.key.get_pressed()
if not pause and not game_over:
    if keys[pygame.K_LEFT] and player_pos[0] > 0:
        player_pos[0] -= player_speed
    if keys[pygame.K_RIGHT] and player_pos[0] < WIDTH - player_size:
        player_pos[0] += player_speed
```

- Checks for continuous key presses (arrow keys) to move the player left or right, considering the game is not paused and not over.

Python Code

```
if not game_over and not pause:
```

```
for enemy in enemies :
        enemy [ 1 ] += enemy_speed
```

- If the game is not over and not paused, the enemies move downward .

Python Code

```
bullets = [[ bullet [ 0 ] , bullet [ 1 ] - bullet_speed ] for bullet in bullets ]
bullets = [ bullet for bullet in bullets if 0 < bullet [ 1 ] < HEIGHT ]
```

- Bullets move upward, and any bullets outside the screen are removed .

Python Code

```
for bullet in bullets [:]:
        for enemy in enemies [:]:
            if (
                enemy [ 0 ] < bullet [ 0 ] < enemy [ 0 ] + enemy_size
                and enemy [ 1 ] < bullet [ 1 ] < enemy [ 1 ] + enemy_size
            ):
                bullets . remove ( bullet )
                enemies . remove ( enemy )
                score += 10
                break
```

- Checks for collisions between bullets and enemies. If a collision occurs, the bullet and enemy are removed, and the score is increased.

Python Code

```
for enemy in enemies :
        if (
            player_pos [ 0 ]  < enemy [ 0 ] +  enemy_size
            and player_pos [ 0 ] +  player_size > enemy [ 0 ]
            and player_pos [ 1 ]  < enemy [ 1 ] +  enemy_size
            and player_pos [ 1 ] +  player_size > enemy [ 1 ]
           ):
            game_over  =  True
            if score > high_score :
                high_score  =  score
            break
```

- Checks for collisions between the player and enemies. If a collision occurs, the game is set to over, and if the score is higher than the previous high score, it is updated.

Python Code

```
if random . randint ( 0, 100 ) < 5 :
        enemy_pos = [ random . randint ( 0, WIDTH - enemy_size ) , 0 ]
        enemies . append ( enemy_pos )
```

- Randomly generates enemies at the top of the screen with a probability of 5 %.

Python Code

```
enemies = [ enemy for enemy in enemies if enemy [ 1 ] < HEIGHT ]
```

- Removes enemies that have gone beyond the bottom of the screen .

Python Code

```
# Draw
screen . fill ( WHITE )
```

- Fills the screen with a white background .

Python Code

```
pygame . draw . rect ( screen, player_color,
                       ( player_pos [ 0 ] , player_pos [ 1 ] , player_size, player_size ))
```

- Draws the player on the screen .

Python Code

```
for tower in towers :
```

```
pygame . draw . rect ( screen, tower_color,
                       ( tower [ 0 ] , tower [ 1 ] , tower_size, tower_size ))
```

- Draws towers on the screen .

Python Code

```
for bullet in bullets :
    pygame . draw . circle ( screen, bullet_color, ( int (
            bullet [ 0 ]) , int ( bullet [ 1 ])) , bullet_size )
```

- Draws bullets on the screen .

Python Code

```
for enemy in enemies :
    pygame . draw . rect ( screen, enemy_color,
                           ( enemy [ 0 ] , enemy [ 1 ] , enemy_size, enemy_size ))
```

- Draws enemies on the screen .

Python Code

```
score_text = font . render ( f " Score : {score} " , True, ( 0, 0, 0 ))
screen . blit ( score_text, ( 10, 10 ))
```

- Renders and displays the current score .

Python Code

```
high_score_text = font.render(f"High Score: {high_score}", True, (0, 0, 0))
screen.blit(high_score_text, (10, 50))
```

- Renders and displays the high score .

Python Code

```
if game_over:
    screen.blit(restart_message, restart_message_rect)
elif pause:
    screen.blit(pause_message, pause_message_rect)
```

- Displays game over or pause messages on the screen, depending on the game state .

Python Code

```
pygame.display.flip()
clock.tick(FPS)
```

- Updates the display and controls the frame rate .

How To Play Tower Defense Game

To play the Tower **Def**ense game described in the provided **Python Code**, follow these instructions :

1. **Objective :**

- The objective of the game is to **def**end your position against waves of incoming enemies .

2. **Controls :**

- Use the **left and right arrow keys** to move the player left and right across the bottom of the screen .
- Press the **spacebar** to shoot bullets upward and destroy incoming enemies .

3. **Towers :**

- There is a tower element in the game, but its functionality is not fully implemented in the provided code . However, you can extend the code to include tower placement and use for additional **def**ense .

4. **Gameplay :**

- Enemies will spawn randomly at the top of the screen and move downwards .
- Your goal is to shoot bullets to destroy the enemies before they reach your position at the bottom of the screen .

5. **Scoring :**

- You earn points for each enemy you successfully eliminate with your bullets .
- The score is displayed on the screen, and there is also a high score that represents your best performance in a single game .

6. **Game Over :**

- The game ends if an enemy collides with your player . In this case, the game over message will be displayed .
- You can restart the game by pressing the 'R' key after a game over . This clears the enemies, bullets, and resets the score .

7. **Pause :**

- You can pause the game by pressing the 'P' key . The pause message will be displayed, and the game will be temporarily halted .
- To resume, press the 'P' key again .

8. **Tips :**

- Try to eliminate enemies efficiently to maximize your score .
- Pay attention to the position of enemies and time your shots strategically .
- Keep an eye on your high score and aim to beat it in each session .

9. **Customization (Optional):**

- You can modify the code to include additional features, such as tower placement and upgrades, more enemy types, and power - ups .

Remember, this game is a basic implementation, and you can enhance it by adding new features and improving the gameplay based on your preferences . Have fun playing and experimenting with the code !

22. Sokuban Game

```
import tkinter as tk
from tkinter import messagebox
import json
```

```
class SokobanGame:
  def __init__(self, master):
    self.master = master
    self.master.title("Sokoban Game")
    self.load_high_score()

    self.levels = [
      {
        'width': 5,
        'height': 5,
        'player_pos': [2, 2],
        'target_pos': [4, 4],
        'box_pos': [3, 3],
      },
      # Add more levels as needed
      {
        'width': 5,
        'height': 5,
        'player_pos': [1, 1],
        'target_pos': [3, 3],
```

```
            'box_pos': [2, 2],
        },
        # ... Add 8 more levels
    ]
    self.current_level = 0
    self.score = 0
    self.hints = 3
    self.create_widgets()

def create_widgets(self):
    self.canvas = tk.Canvas(self.master, width=400, height=400, bg="white")
    self.canvas.grid(row=0, column=0, rowspan=5)

    self.master.bind("<Up>", lambda event: self.move(-1, 0))
    self.master.bind("<Down>", lambda event: self.move(1, 0))
    self.master.bind("<Left>", lambda event: self.move(0, -1))
    self.master.bind("<Right>", lambda event: self.move(0, 1))

    self.restart_button = tk.Button(
        self.master, text="Restart Game", command=self.restart_game)
    self.restart_button.grid(row=0, column=1, sticky="nsew")

    self.hint_button = tk.Button(
        self.master, text="Hint", command=self.show_hint)
```

```
    self.hint_button.grid(row=1, column=1, sticky="nsew")

    self.high_score_label = tk.Label(
        self.master, text=f"High Score: {self.high_score}")
    self.high_score_label.grid(row=2, column=1, sticky="nsew")

    self.load_level()
    self.draw_board()

def load_level(self):
    level_info = self.levels[self.current_level]
    self.width = level_info['width']
    self.height = level_info['height']
    self.player_pos = level_info['player_pos'].copy()
    self.target_pos = level_info['target_pos'].copy()
    self.box_pos = level_info['box_pos'].copy()

def draw_board(self):
    self.canvas.delete("all")

    for row in range(self.height):
        for col in range(self.width):
            x1, y1 = col * 80, row * 80
            x2, y2 = x1 + 80, y1 + 80
```

```
        if [row, col] == self.player_pos:
            self.canvas.create_rectangle(
                x1, y1, x2, y2, fill="blue", outline="black")
        elif [row, col] == self.target_pos:
            self.canvas.create_rectangle(
                x1, y1, x2, y2, fill="green", outline="black")
        elif [row, col] == self.box_pos:
            self.canvas.create_rectangle(
                x1, y1, x2, y2, fill="orange", outline="black")
        else:
            self.canvas.create_rectangle(
                x1, y1, x2, y2, fill="white", outline="black")

    def move(self, dy, dx):
        new_pos = [self.player_pos[0] + dy, self.player_pos[1] + dx]

        if not (0 <= new_pos[0] < self.height and 0 <= new_pos[1] < self.width):
            return

        if new_pos == self.box_pos:
            new_box_pos = [self.box_pos[0] + dy, self.box_pos[1] + dx]

            if not (0 <= new_box_pos[0] < self.height and 0 <= new_box_pos[1] < self.width):
```

```
            return

        self.box_pos = new_box_pos
        self.score += 10  # Increase score when moving the box

    self.player_pos = new_pos
    self.score -= 1  # Decrease score for each move
    self.draw_board()

    if self.check_stuck():
        messagebox.showinfo(
            "Game Over", "You are stuck! Cannot push the box.")
        self.restart_game()

    if self.check_win():
        self.score += 50  # Bonus score for completing the level
        messagebox.showinfo("Congratulations",
                            f"You win!\nYour score: {self.score}")
        self.update_high_score()
        self.next_level()

def check_win(self):
    return self.box_pos == self.target_pos
```

```
def check_stuck(self):
    # Check if the box is stuck in the corners
    corners = [
        [0, 0], [0, self.width - 1],  # Top-left, Top-right corners
        # Bottom-left, Bottom-right corners
        [self.height - 1, 0], [self.height - 1, self.width - 1]
    ]

    for corner in corners:
        if self.box_pos == corner and self.box_pos != self.target_pos:
            return True  # Box is stuck in one of the corners without a green target

    return False  # Box is not stuck

def is_clear(self, position):
    return position == self.target_pos or position != self.player_pos and position != self.box_pos

def next_level(self):
    self.current_level += 1
    if self.current_level < len(self.levels):
        self.load_level()
        self.draw_board()
    else:
        messagebox.showinfo(
```

```
            "Game Over", f"All levels completed!\nFinal score: {self.score}")
        self.restart_game()

    def restart_game(self):
        self.current_level = 0
        self.score = 0
        self.load_level()
        self.draw_board()

    def show_hint(self):
        if self.hints > 0:
            messagebox.showinfo(
                "Hint", "Try to push the box onto the green target!")
            self.hints -= 1
        else:
            messagebox.showinfo(
                "Out of Hints", "You've used all available hints.")

    def load_high_score(self):
        try:
            with open("high_score.json", "r") as file:
                data = json.load(file)
                self.high_score = data.get("high_score", 0)
        except FileNotFoundError:
```

```
        self.high_score = 0

    def update_high_score(self):
        if self.score > self.high_score:
            self.high_score = self.score
            with open("high_score.json", "w") as file:
                json.dump({"high_score": self.high_score}, file)
            self.high_score_label.config(text=f"High Score: {self.high_score}")
```

```
if __name__ == "__main__":
    root = tk.Tk()
    game = SokobanGame(root)
    root.mainloop()
```

let's go through the code line by line to understand its functionality :

Python Code

```
import tkinter as tk
from tkinter import messagebox
import json
```

The code starts by importing the necessary modules : tkinter for creating a graphical user interface, messagebox for displaying pop - up messages, and json for handling JSON file operations .

Python Code

```
class SokobanGame :
    def __init__ ( self, master ):
        self . master = master
        self . master . title (" Sokoban Game ")
        self . load_high_score ()
```

Here, a class named SokobanGame is **def**ined . The constructor (__init__) initializes the game, sets up the main window (master) , and loads the high score from a JSON file .

Python Code

```
        self . levels = [
                {
                    'width' : 5,
                    'height' : 5,
                    'player_pos' : [ 2, 2 ] ,
                    'target_pos' : [ 4, 4 ] ,
```

```
            'box_pos' : [ 3, 3 ] ,
            },
            # Add more levels as needed
            {
            'width' :  5,
            'height' :  5,
            'player_pos' : [ 1, 1 ] ,
            'target_pos' : [ 3, 3 ] ,
            'box_pos' : [ 2, 2 ] ,
            },
        #  ...  Add 8 more levels
    ]
```

A list of levels is **def**ined . Each level is represented by a dictionary containing the width, height, player position, target position, and box position .

Python Code

```
    self . current_level  =  0
    self . score  =  0
    self . hints  =  3
```

```
        self . create_widgets ()
```

Game - related variables are initialized, including the current level, score, and hints . The create_widgets method is then called to set up the graphical user interface .

Python Code

```
def create_widgets ( self ):
    self . canvas  =  tk . Canvas ( self . master, width = 400, height = 400, bg =" white ")
    self . canvas . grid ( row = 0, column = 0, rowspan = 5 )
```

A canvas is created to draw the game board, and it is added to the main window using the grid layout .

Python Code

```
    self . master . bind (" <Up> " , lambda event :  self . move (- 1, 0 ))
    self . master . bind (" <Down> " , lambda event :  self . move ( 1, 0 ))
    self . master . bind (" <Left> " , lambda event :  self . move ( 0,  - 1 ))
    self . master . bind (" <Right> " , lambda event :  self . move ( 0, 1 ))
```

Key bindings are set up to handle arrow key presses, connecting them to the move method with specific direction arguments .

Python Code

```
    self . restart_button  =  tk . Button (
            self . master, text =" Restart Game " , command = self . restart_game )
    self . restart_button . grid ( row = 0, column = 1, sticky =" nsew ")
```

A button for restarting the game is created and added to the layout . Its command is set to the restart_game method .

Python Code

```
self . hint_button = tk . Button (
        self . master, text =" Hint " , command = self . show_hint )
self . hint_button . grid ( row = 1, column = 1, sticky =" nsew ")
```

A button for showing hints is created and added to the layout . Its command is set to the show_hint method .

Python Code

```
self . high_score_label = tk . Label (
        self . master, text = f " High Score : {self . high_score} ")
self . high_score_label . grid ( row = 2, column = 1, sticky =" nsew ")
```

A label to display the high score is created and added to the layout .

Python Code

```
self . load_level ()
self . draw_board ()
```

The load_level method is called to initialize the current level, and the draw_board method is called to display the game board .

The code continues with additional methods and functionalities for handling game logic, level progression, and UI interactions . If you have specific questions or if you'd like me to continue explaining a particular part, feel free to ask !

How To Play Sokoban Game

Sokoban is a classic puzzle game where the player needs to push boxes to specific locations (targets) in a warehouse, with the goal of solving each level . Here's a guide on how to play the Sokoban game implemented in the provided code :

1. **Objective :**
    - The main objective is to push all the boxes onto the green target positions .
2. **Controls :**
    - Use the arrow keys (Up, Down, Left, Right) to move the blue player character around the warehouse .
    - The player can only move to empty spaces in the warehouse .
3. **Game Elements :**
    - **Blue Player (You):** Represented by a blue rectangle . This is the character you control .

- **Orange Boxes：** Represented by orange rectangles . These are the boxes you need to move .
- **Green Targets：** Represented by green rectangles . Boxes must be pushed onto these targets to solve the level .

4. **Rules :**

- The player can only push one box at a time .
- Boxes can only be pushed; they cannot be pulled .
- The player cannot walk through boxes or walls .
- The player loses if a box gets stuck in a corner where there is no green target .

5. **Scoring :**

- Your score is initially set to 0 .
- You earn points for pushing boxes (+ 10 points for each push).
- You lose points for each move (- 1 point for each move).
- Bonus points are awarded for completing a level (+ 50 points).

6. **Buttons :**

- **Restart Game：** Resets the game to the first level, clearing the current score .

- **Hint :** Provides a hint on how to solve the level . You start with three hints, and once they are used up, no more hints are available .

7. **High Score :**

- The high score is displayed on the GUI .
- The high score is updated when you complete a level with a higher score .

8. **Game Over :**

- The game ends when all levels are completed, and a final score is displayed .
- The player can choose to restart the game after completion .

9. **Level Progression :**

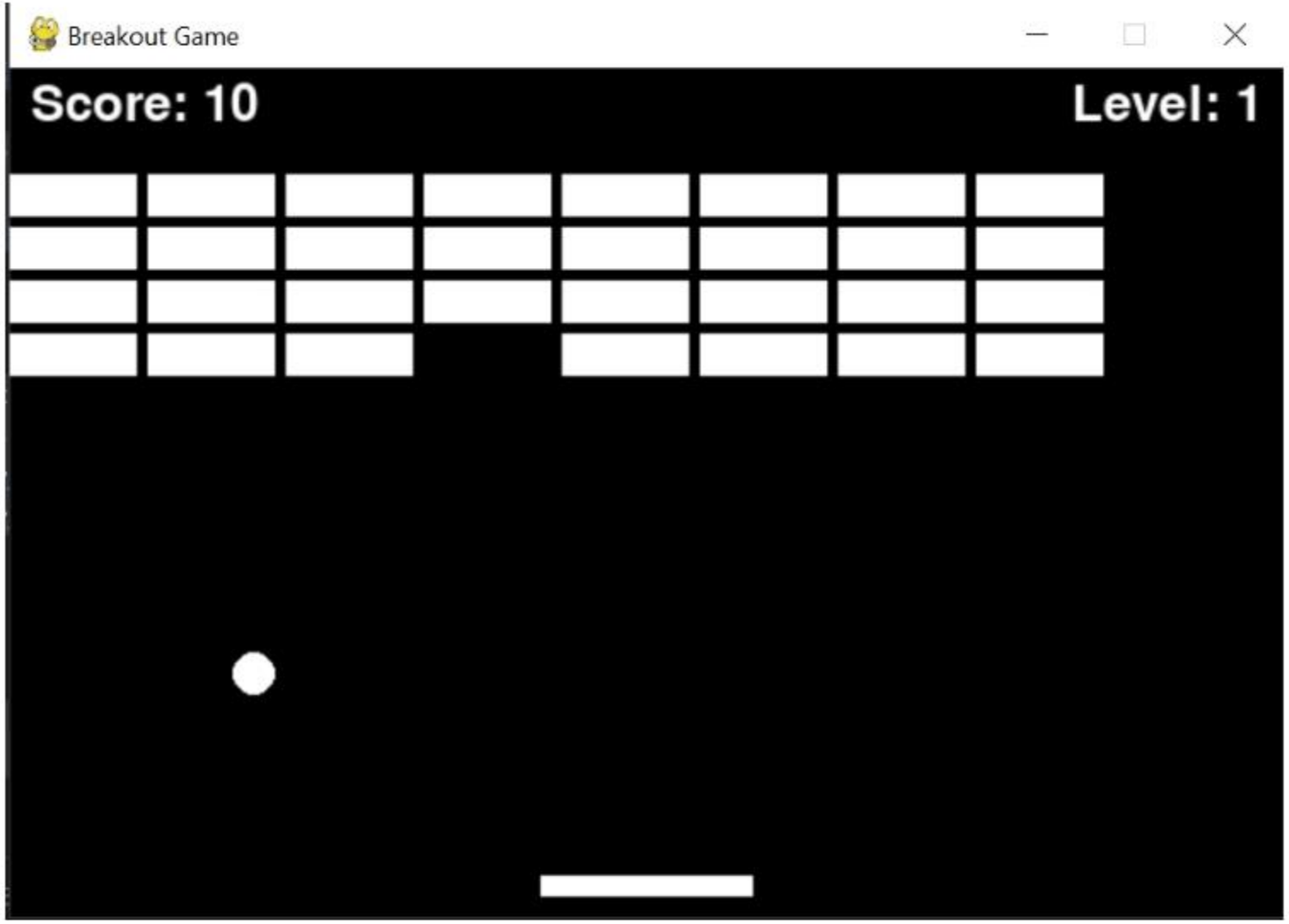
- Successfully pushing all boxes onto the green targets advances you to the next level .
- Completing the last level shows a message indicating that all levels are completed .

10. **Hint Usage :**

- Clicking the " Hint " button provides a hint on how to solve the current level .
- You start with three hints, and the hint count decreases each time you use one .

Remember, Sokoban is a logic puzzle, so take your time to plan your moves and consider the consequences of each action . Good luck and enjoy playing Sokoban !

23. Breakout Game




```
import pygame
import sys
import random

# Initialize Pygame
pygame.init()
```

```
# Constants
WIDTH, HEIGHT = 600, 400
PADDLE_WIDTH, PADDLE_HEIGHT = 100, 10
BALL_RADIUS = 10
BRICK_WIDTH, BRICK_HEIGHT = 60, 20
PADDLE_SPEED = 5
BALL_SPEED = 5
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)

# Sound effects
pygame.mixer.init()
hit_sound = pygame.mixer.Sound("hit.wav")
brick_break_sound = pygame.mixer.Sound("brick_break.wav")
powerup_sound = pygame.mixer.Sound("powerup.wav")

# Create the screen
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Breakout Game")

# Create the paddle
paddle = pygame.Rect(WIDTH // 2 - PADDLE_WIDTH // 2,
            HEIGHT - 20, PADDLE_WIDTH, PADDLE_HEIGHT)
```

```
# Initialize paddle speed and acceleration
paddle_speed = 0

# Create the ball
ball = pygame.Rect(WIDTH // 2 - BALL_RADIUS, HEIGHT // 2 -
            BALL_RADIUS, BALL_RADIUS * 2, BALL_RADIUS * 2)
ball_speed = [random.choice([-1, 1]) * BALL_SPEED, -BALL_SPEED]

# Create bricks
num_bricks_x = 8
num_bricks_y = 4
brick_width = 60
brick_height = 20
bricks = []

for i in range(num_bricks_x):
  for j in range(num_bricks_y):
    brick = pygame.Rect(i * (brick_width + 5), 50 +
                j * (brick_height + 5), brick_width, brick_height)
    bricks.append(brick)

# Game variables
score = 0
```

```
level = 1
game_over = False

# Power-up variables
powerup_active = False
powerup_rect = pygame.Rect(0, 0, 20, 20)
powerup_speed = 3
powerup_duration = 5000  # in milliseconds
powerup_start_time = 0

# Paddle skin options
PADDLE_SKINS = [pygame.Rect(0, 0, 100, 10), pygame.Rect(0, 0, 150, 10)]
paddle_skin_index = 0

# Initialize remaining bricks count
bricks_remaining = num_bricks_x * num_bricks_y

# Main game loop
while True:
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      pygame.quit()
      sys.exit()
    elif event.type == pygame.KEYDOWN and event.key == pygame.K_SPACE and game_over:
```

```
    game_over = False
    score = 0
    level = 1
    ball = pygame.Rect(WIDTH // 2 - BALL_RADIUS, HEIGHT //
                2 - BALL_RADIUS, BALL_RADIUS * 2, BALL_RADIUS * 2)
    ball_speed = [random.choice([-1, 1]) * BALL_SPEED, -BALL_SPEED]

    # Reset bricks
    bricks = []
    for i in range(num_bricks_x):
      for j in range(num_bricks_y):
        brick = pygame.Rect(
          i * (brick_width + 5), 50 + j * (brick_height + 5), brick_width, brick_height)
        bricks.append(brick)

    paddle = pygame.Rect(WIDTH // 2 - PADDLE_WIDTH //
                2, HEIGHT - 20, PADDLE_WIDTH, PADDLE_HEIGHT)
    # Reset remaining bricks count
    bricks_remaining = num_bricks_x * num_bricks_y

if not game_over:
  keys = pygame.key.get_pressed()
  if keys[pygame.K_LEFT] and paddle.left > 0:
    paddle.move_ip(-PADDLE_SPEED, 0)
```

```
if keys[pygame.K_RIGHT] and paddle.right < WIDTH:
    paddle.move_ip(PADDLE_SPEED, 0)

# Update ball position
ball.move_ip(ball_speed[0], ball_speed[1])

# Ball collisions with walls
if ball.left <= 0 or ball.right >= WIDTH:
    ball_speed[0] = -ball_speed[0]
if ball.top <= 0:
    ball_speed[1] = -ball_speed[1]

# Ball collision with paddle
if ball.colliderect(paddle) and ball_speed[1] > 0:
    ball_speed[1] = -ball_speed[1]
    hit_sound.play()

# Ball collisions with bricks
for brick in list(bricks):
    if ball.colliderect(brick):
        bricks.remove(brick)
        bricks_remaining -= 1  # Update remaining bricks count
        ball_speed[1] = -ball_speed[1]
        score += 10
```

```
            brick_break_sound.play()

            # 10% chance to spawn a power-up when a brick is hit
            if random.randint(1, 10) == 1 and not powerup_active:
                powerup_rect.x, powerup_rect.y = brick.x, brick.y
                powerup_active = True
                powerup_start_time = pygame.time.get_ticks()

            # Print remaining bricks count after each removal
            print(f"Remaining bricks: {bricks_remaining}")

    # Ball out of bounds (game over)
    if ball.bottom >= HEIGHT:
        game_over = True
        bricks_remaining = 0  # Reset remaining bricks count

        # Check if there are remaining bricks and stop the sound
        if any(bricks):
            brick_break_sound.stop()

    # Check for level completion
    if bricks_remaining == 0 and not any(bricks):
        level += 1
        ball = pygame.Rect(WIDTH // 2 - BALL_RADIUS, HEIGHT //
```

```
            2 - BALL_RADIUS, BALL_RADIUS * 2, BALL_RADIUS * 2)
ball_speed = [random.choice([-1, 1]) * BALL_SPEED, -BALL_SPEED]
bricks = []
for i in range(num_bricks_x):
    for j in range(num_bricks_y):
        brick = pygame.Rect(
            i * (brick_width + 5), 50 + j * (brick_height + 5), brick_width, brick_height)
        bricks.append(brick)
powerup_active = False  # Reset power-up status
bricks_remaining = num_bricks_x * num_bricks_y  # Reset remaining bricks count

# Ensure the sound and level increase only if there were bricks remaining
if not any(bricks):
    brick_break_sound.stop()  # Stop the sound if it's playing
    level += 1

game_over = False  # Reset game-over state

# Reset paddle position
paddle.x = WIDTH // 2 - PADDLE_WIDTH // 2

# Print remaining bricks count after reset
bricks_remaining = num_bricks_x * num_bricks_y
print(f"Remaining bricks: {bricks_remaining}")
```

```
    # Update power-up position and check its effects
    if powerup_active:
        powerup_rect.y += powerup_speed

        # Check if power-up is collected by the paddle
        if powerup_rect.colliderect(paddle):
            powerup_active = False
            powerup_sound.play()
            # Implement the effect of the power-up (e.g., increase paddle size)
            paddle.width = PADDLE_SKINS[paddle_skin_index].width

        # Check if power-up duration has expired
        if pygame.time.get_ticks() - powerup_start_time > powerup_duration:
            powerup_active = False
            # Reset paddle size
            paddle.width = PADDLE_WIDTH

        # Check if power-up goes out of bounds
        if powerup_rect.top > HEIGHT:
            powerup_active = False

# Draw everything
screen.fill(BLACK)
```

```
# Draw power-up
if powerup_active:
    pygame.draw.rect(screen, (255, 0, 0), powerup_rect)

# Draw paddle
pygame.draw.rect(screen, WHITE, paddle)

# Draw ball
pygame.draw.ellipse(screen, WHITE, ball)

# Draw bricks
for brick in bricks:
    pygame.draw.rect(screen, WHITE, brick)

# Draw score and level
font = pygame.font.Font(None, 36)
score_text = font.render(f"Score: {score}", True, WHITE)
level_text = font.render(f"Level: {level}", True, WHITE)

# Draw background rectangles
pygame.draw.rect(screen, BLACK, (0, 0, WIDTH, score_text.get_height() + 5))
pygame.draw.rect(screen, BLACK, (WIDTH - level_text.get_width() -
                 5, 0, WIDTH, level_text.get_height() + 5))
```

```
# Draw score and level messages
screen.blit(score_text, (10, 5))
screen.blit(level_text, (WIDTH - level_text.get_width() - 10, 5))

# Draw game over screen
if game_over:
  game_over_text = font.render(
    "Game Over! Press SPACE to restart.", True, WHITE)
  screen.blit(game_over_text, (WIDTH // 2 -
        game_over_text.get_width() // 2, HEIGHT // 2))

# Update the display
pygame.display.flip()

# Control the game speed
pygame.time.Clock().tick(60)
```

Let's go through the code line by line to understand its functionality :

Python Code

```
import pygame
import sys
import random
```

1. Import necessary libraries : pygame for creating games, sys for system - related functions, and random for random number generation .

Python Code

```
# Initialize Pygame
pygame . init ()
```

2. Initialize Pygame to set up the gaming environment .

Python Code

```
# Constants
WIDTH, HEIGHT = 600, 400
PADDLE_WIDTH, PADDLE_HEIGHT = 100, 10
BALL_RADIUS = 10
BRICK_WIDTH, BRICK_HEIGHT = 60, 20
PADDLE_SPEED = 5
BALL_SPEED = 5
WHITE = ( 255, 255, 255 )
BLACK = ( 0, 0, 0 )
```

3. **Def**ine constants for various aspects of the game, such as window dimensions, paddle and ball sizes, brick dimensions, speeds, and color codes .

Python Code

```
# Sound effects
pygame . mixer . init ()
hit_sound = pygame . mixer . Sound (" hit . wav ")
brick_break_sound = pygame . mixer . Sound (" brick_break . wav ")
powerup_sound = pygame . mixer . Sound (" powerup . wav ")
```

4. Initialize Pygame's sound mixer and load sound effects for collisions and power - ups .

Python Code

```
# Create the screen
screen = pygame . display . set_mode (( WIDTH, HEIGHT ))
pygame . display . set_caption (" Breakout Game ")
```

5. Set up the game window with the specified dimensions and title .

Python Code

```
# Create the paddle
paddle = pygame . Rect ( WIDTH // 2 - PADDLE_WIDTH // 2, HEIGHT - 20,
PADDLE_WIDTH, PADDLE_HEIGHT )
```

6. Initialize the paddle's position and dimensions using the pygame . Rect class .

Python Code

```
# Initialize paddle speed and acceleration
paddle_speed = 0
```

7. Set the initial paddle speed .

Python Code

```
# Create the ball
ball = pygame . Rect ( WIDTH // 2 - BALL_RADIUS, HEIGHT // 2 - BALL_RADIUS,
BALL_RADIUS * 2, BALL_RADIUS * 2 )
ball_speed = [ random . choice ([- 1, 1 ]) * BALL_SPEED, - BALL_SPEED ]
```

8. Initialize the ball's position, dimensions, and initial speed .

Python Code

```
# Create bricks
num_bricks_x = 8
num_bricks_y = 4
bricks = []

for i in range ( num_bricks_x ):
```

```
    for j in range ( num_bricks_y ):
        brick = pygame . Rect ( i * ( brick_width + 5 ), 50 + j * ( brick_height + 5 ),
brick_width, brick_height )
        bricks . append ( brick )
```

9. Create a grid of bricks using nested loops and store them in a list .

Python Code

```
# Game variables
score = 0
level = 1
game_over = False
```

10. Initialize game variables, including score, level, and game - over status .

Python Code

```
# Power - up variables
powerup_active = False
powerup_rect = pygame . Rect ( 0, 0, 20, 20 )
powerup_speed = 3
powerup_duration = 5000 # in milliseconds
powerup_start_time = 0
```

11. Initialize power - up - related variables, including its status, position, speed, duration, and start time .

Python Code

```
# Paddle skin options
PADDLE_SKINS = [ pygame . Rect ( 0, 0, 100, 10 ) , pygame . Rect ( 0, 0, 150, 10 )]
paddle_skin_index = 0
```

12. **Def**ine different paddle skins using pygame . Rect and set the initial index .

Python Code

```
# Initialize remaining bricks count
bricks_remaining = num_bricks_x * num_bricks_y
```

13. Calculate and initialize the total number of remaining bricks .

Python Code

```
# Main game loop
while True :
    for event in pygame . event . get ():
        if event . type == pygame . QUIT :
                pygame . quit ()
                sys . exit ()
        elif event . type == pygame . KEYDOWN and event . key == pygame . K_SPACE and
game_over :
                # Reset the game if the user presses space after the game is over
                game_over = False
                score = 0
                level = 1
                ball = pygame . Rect ( WIDTH // 2 - BALL_RADIUS, HEIGHT // 2 -
BALL_RADIUS, BALL_RADIUS * 2, BALL_RADIUS * 2 )
                ball_speed = [ random . choice ([- 1, 1 ]) * BALL_SPEED, - BALL_SPEED ]

                # Reset bricks
                bricks = []
                for i in range ( num_bricks_x ):
```

```
            for j in range ( num_bricks_y ):
                brick = pygame . Rect (
                    i * ( brick_width + 5 ) , 50 + j * ( brick_height + 5 ) , brick_width,
brick_height )
                bricks . append ( brick )

        paddle = pygame . Rect ( WIDTH // 2 - PADDLE_WIDTH // 2, HEIGHT - 20,
PADDLE_WIDTH, PADDLE_HEIGHT )
        # Reset remaining bricks count
        bricks_remaining = num_bricks_x * num_bricks_y
```

14. Start the main game loop . Handle events such as quitting the game or restarting it when the space key is pressed after the game over .

15. Inside the game loop, check for user input to quit the game or restart it after a game over .

16. If the game is restarted, reset various game variables, including the score, level, ball, bricks, paddle, and remaining bricks count .

Python Code

```
if not game_over :
```

```
keys = pygame . key . get_pressed ()
if keys [ pygame . K_LEFT ] and paddle . left > 0 :
        paddle . move_ip (- PADDLE_SPEED, 0 )
if keys [ pygame . K_RIGHT ] and paddle . right < WIDTH :
        paddle . move_ip ( PADDLE_SPEED, 0 )
```

17. If the game is not over, check for left and right arrow key presses to move the paddle accordingly .

Python Code

```
  # Update ball position
ball . move_ip ( ball_speed [ 0 ] , ball_speed [ 1 ])
```

18. Update the ball's position based on its current speed .

Python Code

```
  # Ball collisions with walls
if ball . left < = 0 or ball . right > = WIDTH :
        ball_speed [ 0 ] = - ball_speed [ 0 ]
if ball . top < = 0 :
        ball_speed [ 1 ] = - ball_speed [ 1 ]
```

19. Check for collisions with the walls and update the ball's speed accordingly .

Python Code

```
    # Ball collision with paddle
if ball . colliderect ( paddle ) and ball_speed [ 1 ] > 0 :
        ball_speed [ 1 ] = - ball_speed [ 1 ]
        hit_sound . play ()
```

20. Check for collisions with the paddle and update the ball's speed while playing a sound effect .

Python Code

```
    # Ball collisions with bricks
for brick in list ( bricks ):
        if ball . colliderect ( brick ):
            bricks . remove ( brick )
            bricks_remaining -= 1 # Update remaining bricks count
            ball_speed [ 1 ] = - ball_speed [ 1 ]
            score += 10
            brick_break_sound . play ()

            # 10 % chance to spawn a power - up when a brick is hit
            if random . randint ( 1, 10 ) == 1 and not powerup_active :
                powerup_rect . x, powerup_rect . y = brick . x, brick . y
                powerup_active = True
```

```
            powerup_start_time = pygame.time.get_ticks()

        # Print remaining bricks count after each removal
        print(f"Remaining bricks: {bricks_remaining}")
```

21. Check for collisions with bricks, remove the hit bricks, update the remaining bricks count, and play sound effects. Additionally, there's a chance to spawn a power-up when a brick is hit.

22. If a power-up is spawned, set its position and activate it.

23. Print the remaining bricks count after each removal for debugging purposes.

Python Code

```
# Ball out of bounds (game over)
if ball.bottom >= HEIGHT:
        game_over = True
        bricks_remaining = 0 # Reset remaining bricks count

        # Check if there are remaining bricks and stop the sound
        if any(bricks):
            brick_break_sound.stop()
```

24. Check if the ball goes out of bounds (hits the bottom) , trigger a game over, and reset the remaining bricks count . If there are remaining bricks, stop the brick breaking sound .

Python Code

```
    # Check for level completion
    if bricks_remaining == 0 and not any ( bricks ):
        level += 1
        ball = pygame . Rect ( WIDTH // 2 - BALL_RADIUS, HEIGHT // 2 - BALL_RADIUS, BALL_RADIUS * 2, BALL_RADIUS * 2 )
        ball_speed = [ random . choice ([- 1, 1 ]) * BALL_SPEED, - BALL_SPEED ]
        bricks = []
        for i in range ( num_bricks_x ):
            for j in range ( num_bricks_y ):
                brick = pygame . Rect (
                    i * ( brick_width + 5 ) , 50 + j * ( brick_height + 5 ) , brick_width, brick_height )
                bricks . append ( brick )
        powerup_active = False # Reset power - up status
```

```
            bricks_remaining = num_bricks_x * num_bricks_y # Reset remaining bricks
count

            # Ensure the sound and level increase only if there were bricks remaining
            if not any ( bricks ):
                brick_break_sound . stop () # Stop the sound if it's playing
                level += 1

            game_over = False # Reset game - over state

            # Reset paddle position
            paddle . x = WIDTH // 2 - PADDLE_WIDTH // 2

            # Print remaining bricks count after reset
            bricks_remaining = num_bricks_x * num_bricks_y
            print ( f " Remaining bricks : {bricks_remaining} ")
```

25. Check for level completion by verifying if there are no remaining bricks . If so, increment the level, reset various game variables, and restart the level with a new set of bricks .

26. Ensure that the brick breaking sound is stopped only if there were bricks remaining .

27. Reset the game - over state, reset the paddle position, and print the remaining bricks count after the reset for debugging .

Python Code

```
# Update power - up position and check its effects
if powerup_active :
    powerup_rect . y  +=  powerup_speed

    # Check if power - up is collected by the paddle
    if powerup_rect . colliderect ( paddle ):
        powerup_active  =  False
        powerup_sound . play ()
        # Implement the effect of the power - up  ( e . g . , increase paddle size )
        paddle . width  =  PADDLE_SKINS [ paddle_skin_index ]. width

    # Check if power - up duration has expired
    if pygame . time . get_ticks () -  powerup_start_time > powerup_duration :
        powerup_active  =  False
```

```
            # Reset paddle size
        paddle . width  =  PADDLE_WIDTH

    # Check if power - up goes out of bounds
    if powerup_rect . top > HEIGHT :
        powerup_active  =  False
```

28. Update the power - up's position and check its effects, such as collision with the paddle, playing a sound effect, and modifying the paddle's size . Check if the power - up's duration has expired and reset the paddle size accordingly . Also, check if the power - up goes out of bounds .

Python Code

```
 # Draw everything
screen . fill ( BLACK )
```

29. Fill the screen with a black background .

Python Code

```
# Draw power - up
if powerup_active :
    pygame . draw . rect ( screen,  ( 255, 0, 0 ) , powerup_rect )
```

30. If a power - up is active, draw it as a red rectangle on the screen .

Python Code

```
# Draw paddle
pygame . draw . rect ( screen, WHITE, paddle )
```

31. Draw the paddle on the screen .

Python Code

```
# Draw ball
pygame . draw . ellipse ( screen, WHITE, ball )
```

32. Draw the ball on the screen as a white ellipse .

Python Code

```
# Draw bricks
for brick in bricks :
    pygame . draw . rect ( screen, WHITE, brick )
```

33. Draw each brick on the screen as a white rectangle .

Python Code

```
# Draw score and level
font = pygame . font . Font ( None, 36 )
score_text = font . render ( f " Score : {score} " , True, WHITE )
level_text = font . render ( f " Level : {level} " , True, WHITE )
```

34. Create a font object and render the score and level texts in white .

Python Code

```
# Draw background rectangles
pygame . draw . rect ( screen, BLACK, ( 0, 0, WIDTH, score_text . get_height () + 5 ))
pygame . draw . rect ( screen, BLACK, ( WIDTH - level_text . get_width () - 5, 0, WIDTH,
level_text . get_height () + 5 ))
```

35. Draw black rectangles as backgrounds for the score and level texts .

Python Code

```
# Draw score and level messages
screen . blit ( score_text, ( 10, 5 ))
screen . blit ( level_text, ( WIDTH - level_text . get_width () - 10, 5 ))
```

36. Blit the rendered score and level texts onto the screen .

Python Code

```
# Draw game over screen
if game_over :
    game_over_text = font . render (" Game Over ! Press SPACE to restart ." , True,
WHITE )
```

```
    screen . blit ( game_over_text,  ( WIDTH  //  2 -  game_over_text . get_width () //  2,
HEIGHT  //  2 ))
```

37. If the game is over, render and display a game - over message in the center of the screen .

Python Code

```
 # Update the display
pygame . display . flip ()
```

38. Update the display to reflect all the drawing changes .

Python Code

```
 # Control the game speed
pygame . time . Clock (). tick ( 60 )
```

39. Control the game's speed by setting the frame rate to 60 frames per second .

This concludes the line - by - line walkthrough of the Breakout game code .

How To Play Breakout Game

To play the Breakout game, follow these instructions :

1. **Launch the Game :**

- Run the Python script in an environment that supports Pygame .
- The game window will appear with the title " Breakout Game ."

2. **Game Controls :**

- Use the left and right arrow keys on your keyboard to move the paddle horizontally .
- The objective is to bounce the ball off the paddle to hit and break the bricks .

3. **Breaking Bricks :**

- The screen initially contains a grid of bricks at the top .
- Each brick has a certain number of hits required to break it .
- When the ball collides with a brick, the brick disappears, and you earn points .

4. **Power - Ups :**

- Occasionally, hitting a brick may release a power - up .
- If a power - up is released, it will move down the screen .
- Move the paddle to catch the power - up, and it will activate a special ability (e . g . , increasing paddle size).

5. **Level Completion :**

- Your goal is to break all the bricks in the current level .

- When all bricks are broken, you advance to the next level with a new set of bricks .
- The game keeps track of your score and level .

6. **Game Over :**

- If the ball falls below the paddle and touches the bottom of the screen, the game is over .
- You can restart the game by pressing the " SPACE " key after a game over .
- The game will reset, and you can continue playing from the first level .

7. **Score and Level :**

- The score is displayed at the top left corner of the screen .
- The level is displayed at the top right corner of the screen .

8. **Paddle Skins :**

- The game offers different paddle skins that change the appearance of the paddle .
- To switch between paddle skins, you can modify the paddle_skin_index variable in the code .

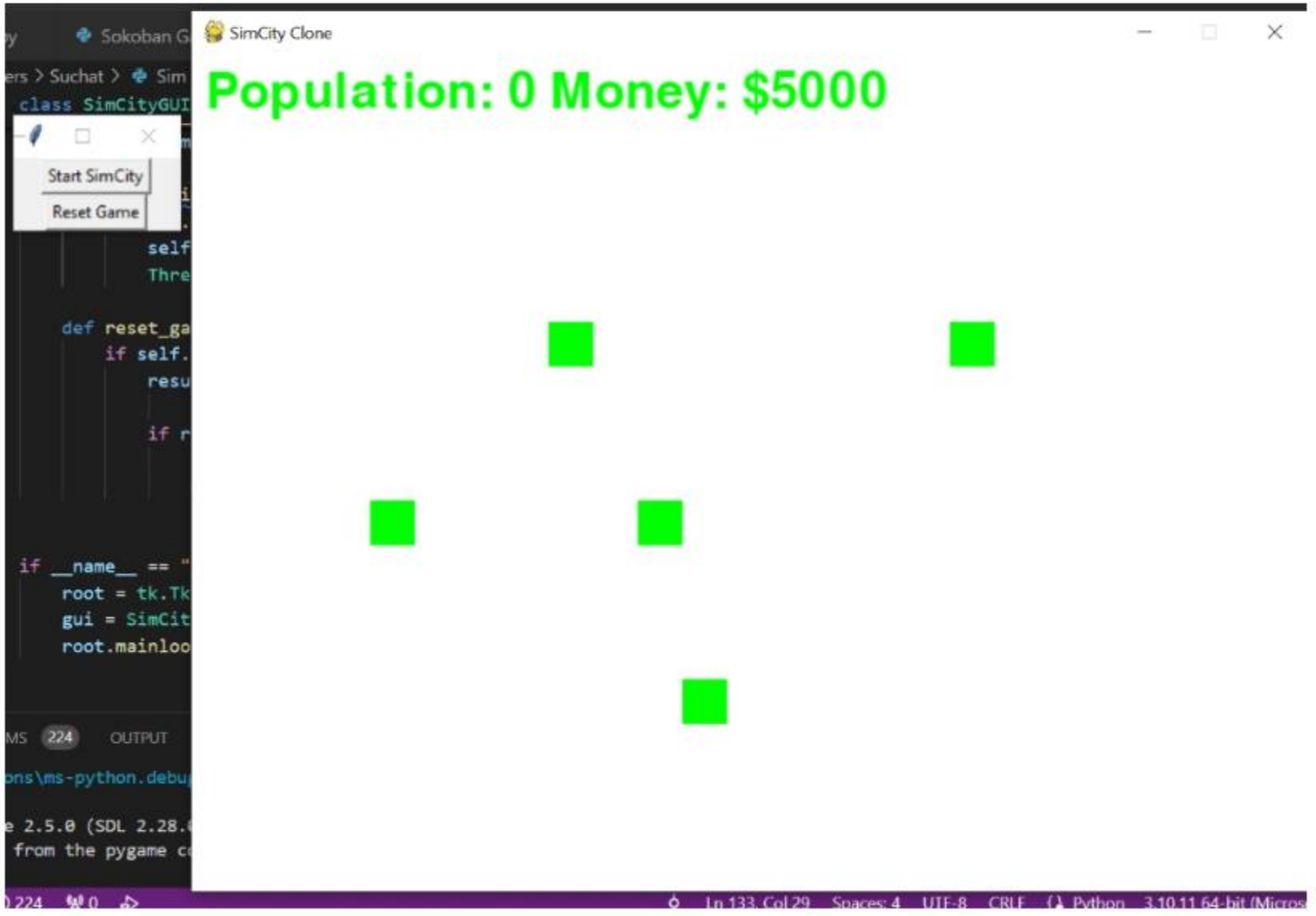
9. **Game Over Message :**

- If the game is over, a " Game Over " message will be displayed in the center of the screen .
- Press the " SPACE " key to restart the game .

10. **Enjoy the Game :**

- Have fun playing the Breakout game and try to achieve the highest score !

24. Sim City Clone Game




```
import pygame
```

```
import sys
import tkinter as tk
from tkinter import messagebox
from threading import Thread

# Constants
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
TILE_SIZE = 32
GRID_WIDTH = SCREEN_WIDTH // TILE_SIZE
GRID_HEIGHT = SCREEN_HEIGHT // TILE_SIZE

# Colors
WHITE = (255, 255, 255)
GREEN = (0, 255, 0)
RESIDENTIAL_COLOR = (0, 0, 255)
```

```
# Zone types
EMPTY = 0

# Services
NO_SERVICE = 0
SCHOOL = 1
HOSPITAL = 2
```

```
POLICE_STATION = 3
PARK = 4
```

```
class SimCity:
  def __init__(self, master):
    self.master = master
    self.master.title("SimCity Clone")

    pygame.init()  # Initialize Pygame
    pygame.font.init()  # Initialize Pygame font module

    self.screen = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT))
    pygame.display.set_caption("SimCity Clone")
    self.grid = [[EMPTY for _ in range(GRID_WIDTH)]
            for _ in range(GRID_HEIGHT)]
    self.population = 0
    self.money = 10000
    self.clock = pygame.time.Clock()
    self.running = False

  def reset_game_state(self):
    # Reset all game-related variables to their initial values
    self.grid = [[EMPTY for _ in range(GRID_WIDTH)]
            for _ in range(GRID_HEIGHT)]
```

```
    self.population = 0
    self.employment = 0
    self.money = 10000
    self.services = [[NO_SERVICE for _ in range(
      GRID_WIDTH)] for _ in range(GRID_HEIGHT)]
    self.pollution = [[0 for _ in range(GRID_WIDTH)]
                for _ in range(GRID_HEIGHT)]
    self.crime = [[0 for _ in range(GRID_WIDTH)]
            for _ in range(GRID_HEIGHT)]
    self.happiness = 100

  def run(self):
    self.running = True
    while self.running:
      for event in pygame.event.get():
        if event.type == pygame.QUIT:
          self.running = False
          self.master.destroy()
          sys.exit()
        elif event.type == pygame.MOUSEBUTTONDOWN:
          self.handle_mouse_click(event.pos)

      self.update()
      self.draw()
```

```
        pygame.display.flip()
        self.clock.tick(60)

    def handle_mouse_click(self, pos):
        col = pos[0] // TILE_SIZE
        row = pos[1] // TILE_SIZE

        # Toggle zone types on mouse click
        if self.grid[row][col] == EMPTY:
            if self.money >= 1000:  # Cost to zone a new area
                self.grid[row][col] = RESIDENTIAL_COLOR

                self.money -= 1000
        # Additional rules for commercial and industrial zones can be added here
```

```
def on_close(self):
    self.running = False
    self.master.destroy()

def update(self):
    # Update logic here
    pass

def draw(self):
```

```
    self.screen.fill(WHITE)
    for row in range(GRID_HEIGHT):
      for col in range(GRID_WIDTH):
        x = col * TILE_SIZE
        y = row * TILE_SIZE
        zone_color = WHITE if self.grid[row][col] == EMPTY else GREEN
        pygame.draw.rect(self.screen, zone_color,
                         (x, y, TILE_SIZE, TILE_SIZE), 0)

    # Display statistics
    font = pygame.font.Font(pygame.font.get_default_font(), 36)
    text = font.render(
      f"Population: {self.population} Money: ${self.money}", True, GREEN)
    self.screen.blit(text, (10, 10))
```

```
class SimCityGUI:
  def __init__(self, master):
    self.master = master
    self.master.title("SimCity GUI")

    self.start_button = tk.Button(
      master, text="Start SimCity", command=self.start_simcity)
    self.start_button.pack()
```

```
    self.reset_button = tk.Button(
        master, text="Reset Game", command=self.reset_game)
    self.reset_button.pack()

    self.game = None

  def start_simcity(self):
    if self.game is None:
      self.game = SimCity(self.master)
      Thread(target=self.game.run).start()

  def reset_game(self):
    if self.game is not None:
      result = messagebox.askquestion(
        "Reset Game", "Are you sure you want to reset the game?")
      if result == 'yes':
        self.game.reset_game_state()  # Use the new method to reset the game state
        messagebox.showinfo("Game Reset", "SimCity has been reset.")


if __name__ == "__main__":
  root = tk.Tk()
  gui = SimCityGUI(root)
```

```
root.mainloop()
```

let's go through the code line by line :

1. import pygame : Imports the Pygame library, which is used for creating games and multimedia applications in Python .
2. import sys : Imports the sys module, which provides access to some variables used or maintained by the Python interpreter and to functions that interact strongly with the interpreter .
3. import tkinter as tk : Imports the tkinter module and renames it as tk . Tkinter is a standard GUI (Graphical User Interface) library for Python .
4. from tkinter import messagebox : Imports the messagebox class from the tkinter module, which is used to display various types of message boxes .
5. from threading import Thread : Imports the Thread class from the threading module, which is used to run the SimCity game in a separate thread to avoid blocking the GUI .
6. **Constants :**
    - SCREEN_WIDTH = 800 : Sets the width of the game screen .
    - SCREEN_HEIGHT = 600 : Sets the height of the game screen .

- TILE_SIZE = 32 : Sets the size of each grid tile .
- GRID_WIDTH = SCREEN_WIDTH // TILE_SIZE : Calculates the number of grid columns based on screen width and tile size .
- GRID_HEIGHT = SCREEN_HEIGHT // TILE_SIZE : Calculates the number of grid rows based on screen height and tile size .

7. **Colors :**

- WHITE , GREEN , RESIDENTIAL_COLOR : RGB tuples representing colors .

8. **Zone types :**

- EMPTY = 0 : Constant representing an empty zone on the grid .

9. **Services :**

- NO_SERVICE = 0 , SCHOOL = 1 , HOSPITAL = 2 , POLICE_STATION = 3 , PARK = 4 : Constants representing different services that can be provided in the city .

10. **Class** SimCity **:**

- __init__ : Initializes the SimCity game instance with various attributes .
- reset_game_state : Resets the game - related variables to their initial values .
- run : The main game loop that handles events, updates the game state, and draws the screen .

- handle_mouse_click : Handles mouse clicks to toggle zone types on the grid .
- on_close : Handles closing the game window .
- update and draw : Placeholder methods for game logic update and screen drawing .

11. **Class** SimCityGUI :

- __init__ : Initializes the SimCity GUI with buttons for starting and resetting the game .
- start_simcity : Creates a new SimCity instance and starts it in a separate thread .
- reset_game : Asks for confirmation before resetting the game state .

12. **Main Block :**

- Creates a tkinter root window (root).
- Initializes a SimCityGUI instance (gui).
- Enters the tkinter main event loop (root . mainloop ()).

This code sets up a basic structure for a SimCity - like game with a GUI using Pygame and tkinter . The game allows zoning different areas on a grid and includes basic functionality for starting, resetting, and interacting with the game . The game logic and additional features can be implemented in the update method and other relevant parts of the code .

How To Play Sim City Clone Game

The SimCity clone game provided in the code is a basic simulation where you can zone residential areas on a grid . Below are the steps on how to play the game :

1. **Start the Game :**
    - Run the provided Python script .
    - A tkinter GUI window will appear with " Start SimCity " and " Reset Game " buttons .
2. **Start SimCity :**
    - Click the " Start SimCity " button .
    - This will create a new Pygame window where you can interact with the game .
3. **Zoning :**
    - In the Pygame window, you'll see an empty grid . Each cell represents a tile .
    - Click on a tile to zone it as a residential area (indicated by a green color).
    - Zoning a new area costs $1000 (as **def**ined in the handle_mouse_click method).
4. **Population and Money :**
    - The top left of the Pygame window displays the current population and available money .

- Population increases when you zone residential areas .
- Money decreases when you zone a new residential area .

5. **Reset Game :**

- If you want to start over, go back to the tkinter GUI window .
- Click the " Reset Game " button .
- A confirmation dialog will appear . Click " Yes " to reset the game .

6. **Close the Game :**

- You can close the game window or the tkinter GUI window at any time .
- Closing the game window will stop the simulation .

7. **Game Logic (Update and Draw):**

- The game logic, including population growth, zoning costs, and other features, can be implemented in the update method of the SimCity class .
- Drawing and displaying information are handled in the draw method .

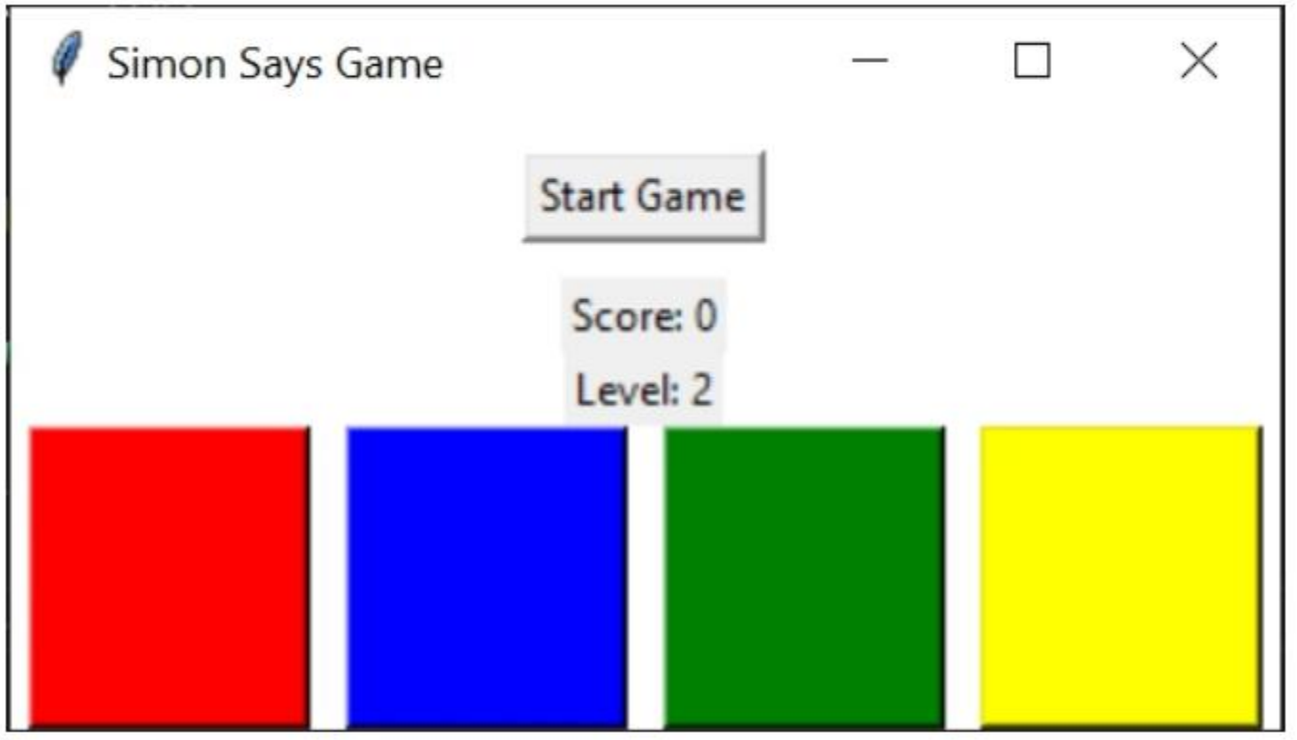
8. **Extend the Game :**

- To make the game more interesting, you can extend the functionality :
  - Add commercial and industrial zones with different costs and effects .

- Implement services such as schools, hospitals, and police stations .
- Introduce pollution and crime factors affecting happiness and population growth .
- Implement a win or lose condition .

Remember that this is a basic starting point, and you can enhance and customize the game according to your preferences by modifying the code in the SimCity class .

25. Simon Says Game




```
import tkinter as tk
import random
import time
import winsound  # For playing sound effects (Windows only)
```

```
class SimonSaysGame:
    def __init__(self, master):
        self.master = master
        self.master.title("Simon Says Game")
```

```
    self.colors = ["red", "blue", "green", "yellow"]
    self.sequence = []
    self.user_sequence = []
    self.round = 1
    self.speed = 1000  # Initial speed in milliseconds
    self.game_running = False

    self.create_widgets()
    self.new_game()

def create_widgets(self):
    self.start_button = tk.Button(
        self.master, text="Start Game", command=self.start_game)
    self.start_button.pack(pady=10)

    self.score_label = tk.Label(self.master, text="Score: 0")
    self.score_label.pack()

    self.level_label = tk.Label(self.master, text="Level: 1")
    self.level_label.pack()

    for color in self.colors:
        button = tk.Button(self.master, bg=color, width=10, height=5,
```

```
                command=lambda c=color: self.check_sequence(c))
        button.pack(side=tk.LEFT, padx=5)

    def new_game(self):
        self.round = 1
        self.speed = 1000
        self.game_running = False
        self.sequence = []
        self.user_sequence = []
        self.start_button.configure(state=tk.NORMAL)
        self.update_score_label()
        self.update_level_label()

    def start_game(self):
        self.new_game()
        self.play_sequence()

    def play_sequence(self):
        self.game_running = True
        self.start_button.configure(state=tk.DISABLED)

        for _ in range(self.round):
            new_color = random.choice(self.colors)
            self.sequence.append(new_color)
```

```
            self.highlight_color(new_color)
            time.sleep(self.speed / 1000)
            self.reset_colors()

        self.prompt_user()

    def highlight_color(self, color):
        self.master.configure(bg=color)
        self.master.update()
        self.play_sound(color)
        time.sleep(self.speed / 2000)
        self.master.update()

    def reset_colors(self):
        self.master.configure(bg="white")
        self.master.update()

    def prompt_user(self):
        self.user_sequence = []

    def check_sequence(self, color):
        if self.game_running:
            self.user_sequence.append(color)
            self.highlight_color(color)
```

```
        self.master.after(500, self.reset_colors)
        if self.user_sequence == self.sequence:
            if len(self.user_sequence) == len(self.sequence):
                self.master.after(500, self.display_correct_feedback)
                self.round += 1
                self.speed -= 20  # Increase speed for the next round
                self.update_score_label()
                self.update_level_label()
                self.master.after(1000, self.play_sequence)
        else:
            self.end_game()

    def display_correct_feedback(self):
        self.master.configure(bg="green")
        self.master.after(500, self.reset_colors)

    def end_game(self):
        self.master.configure(bg="red")
        self.start_button.configure(state=tk.NORMAL)
        self.game_running = False
        self.play_sound("wrong")

    def update_score_label(self):
        self.score_label.config(text=f"Score: {max(0, len(self.sequence) - 1)}")
```

```
def update_level_label(self):
    self.level_label.config(text=f"Level: {self.round}")

def play_sound(self, color):
    # Play sound effects based on the color (Windows only)
    if color == "red":
        winsound.PlaySound("SystemExclamation", winsound.SND_ASYNC)
    elif color == "blue":
        winsound.PlaySound("SystemAsterisk", winsound.SND_ASYNC)
    elif color == "green":
        winsound.PlaySound("SystemQuestion", winsound.SND_ASYNC)
    elif color == "yellow":
        winsound.PlaySound("SystemHand", winsound.SND_ASYNC)
    elif color == "wrong":
        winsound.PlaySound("SystemExit", winsound.SND_ASYNC)

if __name__ == "__main__":
    root = tk.Tk()
    game = SimonSaysGame(root)
    root.mainloop()
```

let's go through the code line by line :

Python Code

```
import tkinter as tk
import random
import time
import winsound  # For playing sound effects ( Windows only )
```

- This block imports the necessary modules : tkinter for GUI, random for generating random colors, time for introducing delays, and winsound for playing sound effects (note that winsound is specific to Windows).

Python Code

```
class SimonSaysGame :
    def __init__ ( self, master ):
        self . master  =  master
        self . master . title (" Simon Says Game ")

        self . colors  = [" red " , " blue " , " green " , " yellow "]
        self . sequence  = []
        self . user_sequence  = []
        self . round =  1
        self . speed  =  1000  # Initial speed in milliseconds
```

```
self . game_running = False

self . create_widgets ()
self . new_game ()
```

- This **def**ines a class SimonSaysGame , which is the main class for the Simon Says game .
- The __init__ method initializes the game by setting up the main window (master), **def**ining color options, initializing various game - related variables, and calling two methods : create_widgets and new_game .

Python Code

```
def create_widgets ( self ):
    self . start_button = tk . Button (
            self . master, text =" Start Game " , command = self . start_game )
    self . start_button . pack ( pady = 10 )

    self . score_label = tk . Label ( self . master, text =" Score : 0 ")
    self . score_label . pack ()

    self . level_label = tk . Label ( self . master, text =" Level : 1 ")
    self . level_label . pack ()
```

```
for color in self . colors :
        button  =  tk . Button ( self . master, bg = color, width = 10, height = 5,
                                command = lambda c = color :  self . check_sequence ( c ))
        button . pack ( side = tk . LEFT, padx = 5 )
```

- The create_widgets method sets up the GUI elements, including a " Start Game " button, score label, level label, and colored buttons for the game . Each colored button has a command associated with the check_sequence method, which is passed the color of the button .

Python Code

```
def new_game ( self ):
    self . round =  1
    self . speed  =  1000
    self . game_running  =  False
    self . sequence  = []
    self . user_sequence  = []
    self . start_button . configure ( state = tk . NORMAL )
    self . update_score_label ()
    self . update_level_label ()
```

- The new_game method resets various game - related variables, configures the " Start Game " button, and updates the score and level labels .

Python Code

```
def start_game ( self ):
    self . new_game ()
    self . play_sequence ()
```

- The start_game method initializes a new game and starts playing the sequence .

Python Code

```
def play_sequence ( self ):
    self . game_running = True
    self . start_button . configure ( state = tk . DISABLED )

    for _ in range ( self . round ):
            new_color = random . choice ( self . colors )
            self . sequence . append ( new_color )
            self . highlight_color ( new_color )
            time . sleep ( self . speed / 1000 )
            self . reset_colors ()
```

```
self . prompt_user ()
```

- The play_sequence method generates and displays a sequence of colors, one color at a time . It disables the " Start Game " button during this process .

Python Code

```
def highlight_color ( self, color ):
    self . master . configure ( bg = color )
    self . master . update ()
    self . play_sound ( color )
    time . sleep ( self . speed / 2000 )
    self . master . update ()
```

- The highlight_color method changes the background color of the main window, plays a sound associated with the color, introduces a delay, and then resets the background color .

Python Code

```
def reset_colors ( self ):
    self . master . configure ( bg =" white ")
    self . master . update ()
```

- The reset_colors method resets the background color to white .

Python Code

```
def prompt_user ( self ):
    self . user_sequence  = []
```

- The prompt_user method resets the user's sequence .

Python Code

```
def check_sequence ( self, color ):
    if self . game_running :
            self . user_sequence . append ( color )
            self . highlight_color ( color )
            self . master . after ( 500, self . reset_colors )
            if self . user_sequence  ==  self . sequence :
                if len ( self . user_sequence ) ==  len ( self . sequence ):
                    self . master . after ( 500, self . display_correct_feedback )
                    self . round += 1
                    self . speed  -= 20  # Increase speed for the next round
                    self . update_score_label ()
                    self . update_level_label ()
                    self . master . after ( 1000, self . play_sequence )
```

```
        else :
            self . end_game ()
```

- The check_sequence method is called when a colored button is pressed . It adds the color to the user's sequence, highlights the color, resets the color, and checks if the user's sequence matches the generated sequence . If it matches, it updates the score, level, and continues to the next round . If it doesn't match, the game ends .

Python Code

```
def display_correct_feedback ( self ):
    self . master . configure ( bg =" green ")
    self . master . after ( 500, self . reset_colors )
```

- The display_correct_feedback method briefly changes the background color to green to provide feedback when the user correctly matches the sequence .

Python Code

```
def end_game ( self ):
    self . master . configure ( bg =" red ")
    self . start_button . configure ( state = tk . NORMAL )
    self . game_running  =  False
    self . play_sound (" wrong ")
```

- The end_game method changes the background color to red, enables the " Start Game " button, sets the game status to not running, and plays a " wrong " sound .

Python Code

```
def update_score_label ( self ):
    self . score_label . config ( text = f " Score :  {max ( 0, len ( self . sequence ) -  1 ) } ")
```

- The update_score_label method updates the score label based on the length of the sequence .

Python Code

```
def update_level_label ( self ):
    self . level_label . config ( text = f " Level :  {self . round} ")
```

- The update_level_label method updates the level label based on the current round .

Python Code

```
def play_sound ( self, color ):
```

```
    # Play sound effects based on the color ( Windows only )
    if color == " red ":
            winsound . PlaySound (" SystemExclamation " , winsound . SND_ASYNC )
    elif color == " blue ":
            winsound . PlaySound (" SystemAsterisk " , winsound . SND_ASYNC )
    elif color == " green ":
            winsound . PlaySound (" SystemQuestion " , winsound . SND_ASYNC )
    elif color == " yellow ":
            winsound . PlaySound (" SystemHand " , winsound . SND_ASYNC )
    elif color == " wrong ":
            winsound . PlaySound (" SystemExit " , winsound . SND_ASYNC )
```

- The play_sound method plays sound effects based on the color using the winsound module .

Python Code

```
if __name__ == " __main__ ":
    root = tk . Tk ()
    game = SimonSaysGame ( root )
    root . mainloop ()
```

- Finally, this block creates an instance of the `SimonSaysGame` class and starts the main loop using `root.mainloop()`.

How To Play Simon Says Game

To play the Simon Says game, follow these steps :

1. **Run the Code :**
    - Execute the provided **Python Code** in an environment that supports GUI applications and the `tkinter` library.
    - Ensure that you are running the code on a Windows system, as it uses the `winsound` module for sound effects, which is specific to Windows.
2. **Game Interface :**
    - After running the code, a window titled "Simon Says Game" will appear.
3. **Start the Game :**
    - Click the "Start Game" button to initiate the game.
4. **Observe the Sequence :**
    - The game will generate a sequence of colored buttons, and each button will be highlighted one at a time.

- Pay close attention to the sequence, as you will need to replicate it .

5. **Repeat the Sequence :**

- After the sequence is displayed, the background color of the window changes to white .
- Click on the colored buttons in the same order as they were highlighted in the sequence .
- The program will provide visual and auditory feedback for each correct button press .

6. **Advance to the Next Level :**

- If you successfully repeat the sequence, the game will proceed to the next level .
- The level and score will be updated accordingly .

7. **Speed Increase :**

- As you progress through levels, the speed of the sequence display will increase, making the game more challenging .

8. **End of Game :**

- If you make a mistake and press a button out of sequence, the game will end .
- The background color will turn red, and a " wrong " sound effect will be played .

9. **Restart the Game :**

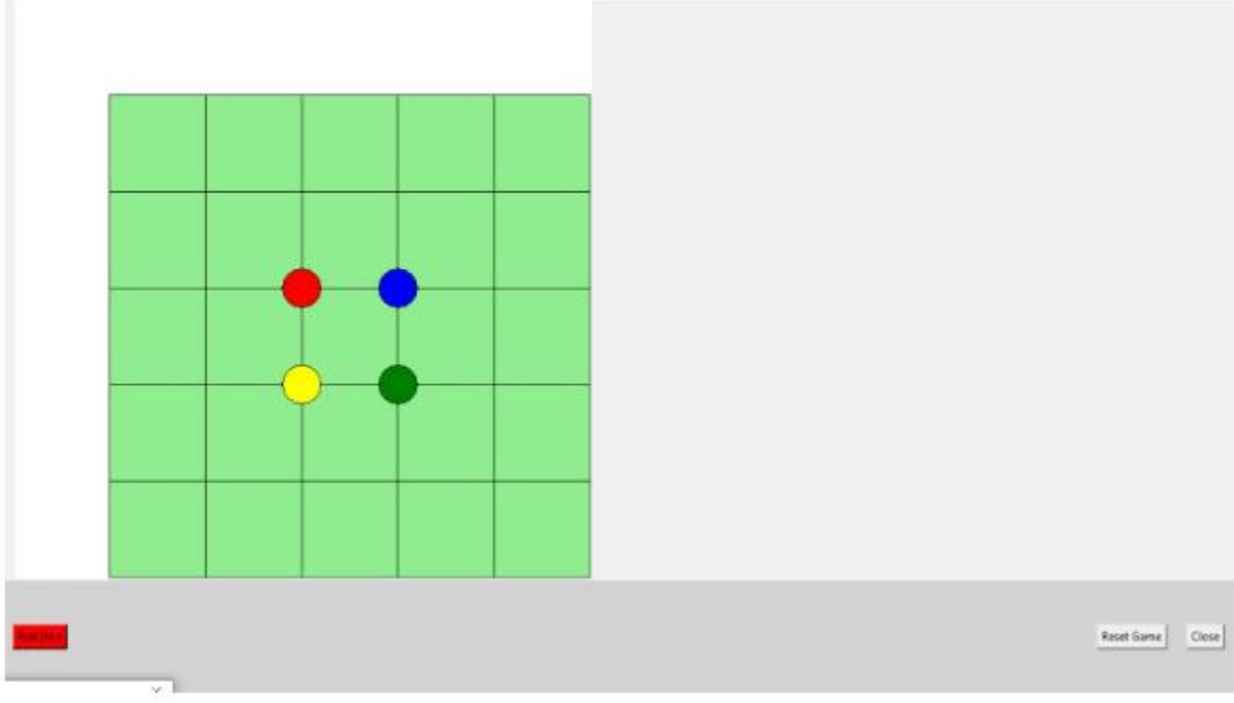
- You can restart the game by clicking the " Start Game " button again .

10. **Enjoy and Improve :**

- The goal is to reach the highest level possible by accurately repeating the growing sequences .
- Challenge yourself to improve your memory and reaction time .

Please note that the code assumes you are using a Windows system for sound effects . If you are on a different operating system, you may need to modify the play_sound function to use a cross - platform sound library .

26. Ludo Game



```
import tkinter as tk
import random
import time
```

```
class LudoGame:
  def __init__(self, root):
    self.root = root
    self.root.title("Ludo Game")
    self.root.attributes('-fullscreen', True)

    # Add background color for buttons
    button_frame = tk.Frame(root, bg="lightgray")
    button_frame.pack(side=tk.BOTTOM, fill=tk.BOTH, expand=True)

    self.canvas = tk.Canvas(root, width=600, height=600, bg="white")
    self.canvas.pack(side=tk.LEFT, padx=10)

    self.create_board()
    self.create_players()

    # Create Roll Dice button
    self.roll_button = tk.Button(
      button_frame, text="Roll Dice", command=self.play_turn)
    self.roll_button.pack(side=tk.LEFT, padx=10, pady=10)
```

```
    # Create Close button
    self.close_button = tk.Button(
        button_frame, text="Close", command=root.destroy)
    self.close_button.pack(side=tk.RIGHT, padx=10, pady=10)

    # Create Reset Game button
    self.reset_button = tk.Button(
        button_frame, text="Reset Game", command=self.reset_game)
    self.reset_button.pack(side=tk.RIGHT, padx=10, pady=10)

    # Create label to display player information
    self.player_info_label = tk.Label(
        root, text="", font=("Helvetica", 16))
    self.player_info_label.pack(side=tk.TOP, pady=10)

    # Initialize variables for current player and piece
    self.current_player_index = 0

    # Highlight the current player's turn
    self.highlight_current_player()

def create_board(self):
    # Draw the Ludo board
```

```
    for i in range(1, 6):
        for j in range(1, 6):
            x1, y1 = i * 100, j * 100
            x2, y2 = x1 + 100, y1 + 100
            self.canvas.create_rectangle(
                x1, y1, x2, y2, outline="black", fill="lightgreen")

def create_players(self):
    self.players = [
        {"color": "red", "piece": {"position": (300, 300)}},
        {"color": "green", "piece": {"position": (400, 400)}},
        {"color": "blue", "piece": {"position": (400, 300)}},
        {"color": "yellow", "piece": {"position": (300, 400)}}
    ]

    for player in self.players:
        self.draw_piece(player["piece"]["position"], player["color"])

def draw_piece(self, position, color):
    x, y = position
    self.canvas.create_oval(
        x - 20, y - 20, x + 20, y + 20, outline="black", fill=color, tags="piece")

def roll_dice(self):
```

```
    # Simulate a dice animation
    for _ in range(10):
        value = random.randint(1, 6)
        self.roll_button.config(text=value)
        self.root.update()
        time.sleep(0.1)

    # Display the final dice value
    final_value = random.randint(1, 6)
    self.roll_button.config(text=final_value)
    return final_value

def move_piece(self, steps):
    player = self.players[self.current_player_index]
    current_position = player["piece"]["position"]

    # Calculate the new position based on the steps rolled
    new_position = self.calculate_new_position(current_position, steps)

    # Check if the new position is valid
    if self.is_valid_position(new_position):
        self.clear_position(current_position)
        player["piece"]["position"] = new_position
        self.draw_piece(new_position, player["color"])
```

```
        # Check for win condition
        if self.check_win_condition():
            winner = player['color']
            self.player_info_label.config(text=f"{winner} player wins!")

        # Highlight the current player's turn
        self.highlight_current_player()

    def calculate_new_position(self, current_position, steps):
        x, y = current_position
        x += steps * 20
        return x, y

    def is_valid_position(self, position):
        x, y = position
        return 0 < x < 600 and 0 < y < 600

    def clear_position(self, position):
        x, y = position
        overlapping_items = self.canvas.find_overlapping(
            x - 20, y - 20, x + 20, y + 20)
        for item in overlapping_items:
            tags = self.canvas.gettags(item)
```

```
            if "piece" in tags:
                self.canvas.delete(item)

    def play_turn(self):
        steps = self.roll_dice()

        # Update player information label
        player = self.players[self.current_player_index]
        self.player_info_label.config(
            text=f"{player['color']} player's turn - Dice Roll: {steps}")

        # Move the piece based on the dice roll
        self.move_piece(steps)

        # Update current player index
        self.current_player_index += 1
        if self.current_player_index >= len(self.players):
            self.current_player_index = 0

    def highlight_current_player(self):
        # Reset background color for all players
        for player in self.players:
            self.roll_button.config(
                bg="SystemButtonFace")  # Reset button color
```

```
    # Highlight the background of the current player's roll dice button
    current_player = self.players[self.current_player_index]
    color = current_player["color"]
    self.roll_button.config(bg=color)

def reset_game(self):
    # Clear all pieces from the canvas
    self.canvas.delete("piece")

    # Reset player positions
    for player in self.players:
        player["piece"]["position"] = self.get_start_position(
            player["color"])
        self.draw_piece(player["piece"]["position"], player["color"])

    # Reset player information label
    self.player_info_label.config(text="Game Reset")

    # Reset current player index
    self.current_player_index = 0

    # Highlight the current player's turn
    self.highlight_current_player()
```

```
  def get_start_position(self, color):
    # Return the starting position for the given player color
    if color == "red":
      return (300, 300)
    elif color == "green":
      return (400, 400)
    elif color == "blue":
      return (400, 300)
    elif color == "yellow":
      return (300, 400)

  def check_win_condition(self):
    # Check if a player has reached a certain position (e.g., the center of the board)
    center_position = (300, 300)
    for player in self.players:
      if player['piece']['position'] == center_position:
        return True
    return False
```

```
# Main program
root = tk.Tk()
ludo_game = LudoGame(root)
root.mainloop()
```

let's go through the code line by line to understand its functionality :

Python Code

```
import tkinter as tk
import random
import time
```

This part of the code imports the necessary modules for creating a graphical user interface (GUI) using the Tkinter library . The random module is used for simulating dice rolls, and time is used for adding a delay in the dice animation .

Python Code

```
class LudoGame :
    def __init__ ( self, root ):
```

Here, a class LudoGame is **def**ined, which will represent the Ludo game . The __init__ method is a special method called when an object of the class is created . It initializes the game with the given root (Tkinter root window).

Python Code

```
        self . root = root
        self . root . title (" Ludo Game ")
        self . root . attributes ( ' - fullscreen', True )
```

These lines store the root window, set its title to " Ludo Game, " and make it fullscreen .

Python Code

```
button_frame = tk . Frame ( root, bg =" lightgray ")
button_frame . pack ( side = tk . BOTTOM, fill = tk . BOTH, expand = True )
```

This creates a frame (button_frame) at the bottom of the root window with a light gray background . The frame is set to expand in both horizontal and vertical directions .

Python Code

```
self . canvas = tk . Canvas ( root, width = 600, height = 600, bg =" white ")
self . canvas . pack ( side = tk . LEFT, padx = 10 )
```

This creates a canvas within the root window with a white background, representing the game board . The canvas is set to a fixed size of 600x600 and is packed to the left of the root window with some padding .

Python Code

```
self . create_board ()
self . create_players ()
```

These lines call methods to create the Ludo board (create_board) and initialize player positions (create_players).

Python Code

```
self . roll_button = tk . Button (
```

```
        button_frame, text =" Roll Dice " , command = self . play_turn )
    self . roll_button . pack ( side = tk . LEFT, padx = 10, pady = 10 )
```

This creates a button labeled " Roll Dice " inside the button_frame . Clicking this button triggers the play_turn method .

Python Code

```
    self . close_button = tk . Button (
        button_frame, text =" Close " , command = root . destroy )
    self . close_button . pack ( side = tk . RIGHT, padx = 10, pady = 10 )
```

This creates a " Close " button within the button_frame that, when clicked, closes the Tkinter window .

Python Code

```
    self . reset_button = tk . Button (
        button_frame, text =" Reset Game " , command = self . reset_game )
    self . reset_button . pack ( side = tk . RIGHT, padx = 10, pady = 10 )
```

This creates a " Reset Game " button within the button_frame that, when clicked, triggers the reset_game method .

Python Code

```
    self . player_info_label = tk . Label (
```

```
            root, text ="" , font =(" Helvetica " , 16 ))
        self . player_info_label . pack ( side = tk . TOP, pady = 10 )
```

This creates a label at the top of the root window to display player information . The initial text is empty, and the font is set to " Helvetica " with a size of 16 .

Python Code

```
        self . current_player_index = 0
        self . highlight_current_player ()
```

These lines initialize the variable for the current player index and call a method to highlight the current player's turn .

Python Code

```
    def create_board ( self ):
        for i in range ( 1, 6 ):
                for j in range ( 1, 6 ):
                    x1, y1 = i * 100, j * 100
                    x2, y2 = x1 + 100, y1 + 100
                    self . canvas . create_rectangle (
                        x1, y1, x2, y2, outline =" black " , fill =" lightgreen ")
```

The create_board method draws the Ludo board on the canvas using nested loops . It creates a 5x5 grid of rectangles with black outlines and light green fill .

Python Code

```
def create_players ( self ):
    self . players  = [
            { " color ": " red " , " piece ": { " position ": ( 300, 300 ) }},
            { " color ": " green " , " piece ": { " position ": ( 400, 400 ) }},
            { " color ": " blue " , " piece ": { " position ": ( 400, 300 ) }},
            { " color ": " yellow " , " piece ": { " position ": ( 300, 400 ) }}
    ]

    for player in self . players :
            self . draw_piece ( player [" piece "][" position "] , player [" color "])
```

The create_players method initializes a list of players, each represented by a dictionary with a color and initial piece position . It then calls the draw_piece method to draw each player's piece on the canvas .

Python Code

```
def draw_piece ( self, position, color ):
    x, y  =  position
```

```
self . canvas . create_oval (
        x - 20, y - 20, x + 20, y + 20, outline =" black " , fill = color, tags =" piece ")
```

The draw_piece method is responsible for drawing a player's piece on the canvas . It uses the oval shape to represent the piece, given its position and color . The tags parameter is used to tag the created object as a " piece ."

Python Code

```
def roll_dice ( self ):
    for _ in range ( 10 ):
            value = random . randint ( 1, 6 )
            self . roll_button . config ( text = value )
            self . root . update ()
            time . sleep ( 0 . 1 )

    final_value = random . randint ( 1, 6 )
    self . roll_button . config ( text = final_value )
    return final_value
```

The roll_dice method simulates a dice animation by updating the text on the " Roll Dice " button with random values . It uses the random . randint function and introduces a slight delay between updates to create the rolling effect .

Python Code

```
def move_piece ( self, steps ):
    player = self . players [ self . current_player_index ]
    current_position = player [" piece "][" position "]

    new_position = self . calculate_new_position ( current_position, steps )

    if self . is_valid_position ( new_position ):
            self . clear_position ( current_position )
            player [" piece "][" position "] = new_position
            self . draw_piece ( new_position, player [" color "])

            if self . check_win_condition ():
                winner = player [ 'color' ]
                self . player_info_label . config ( text = f " {winner} player wins !")

            self . highlight_current_player ()
```

The move_piece method handles moving the player's piece based on the steps rolled . It calculates the new position, checks its validity, clears the previous position, updates the player's position, redraws the piece, checks for a win condition, and highlights the next player's turn .

Python Code

```
def calculate_new_position ( self, current_position, steps ):
    x, y = current_position
    x += steps * 20
    return x, y
```

The calculate_new_position method computes the new position based on the current position and the number of steps rolled .

Python Code

```
def is_valid_position ( self, position ):
    x, y = position
    return 0 < x < 600 and 0 < y < 600
```

The is_valid_position method checks if a given position is within the boundaries of the canvas .

Python Code

```
def clear_position ( self, position ):
    x, y = position
    overlapping_items = self . canvas . find_overlapping (
            x - 20, y - 20, x + 20, y + 20 )
```

```
        for item in overlapping_items :
                tags = self . canvas . gettags ( item )
                if " piece " in tags :
                        self . canvas . delete ( item )
```

The clear_position method deletes any items (pieces) present at a given position on the canvas .

Python Code

```
def play_turn ( self ):
    steps = self . roll_dice ()

    player = self . players [ self . current_player_index ]
    self . player_info_label . config (
            text = f " {player [ 'color' ] } player's turn - Dice Roll : {steps} ")

    self . move_piece ( steps )

    self . current_player_index += 1
    if self . current_player_index > = len ( self . players ):
            self . current_player_index = 0
```

The play_turn method initiates a player's turn by rolling the dice, updating the player information label, moving the piece, and advancing to the next player .

Python Code

```
def highlight_current_player ( self ):
    for player in self . players :
            self . roll_button . config (
                bg =" SystemButtonFace ")  # Reset button color
```

```
current_player = self . players [ self . current_player_index ]
color = current_player [" color "]
self . roll_button . config ( bg = color )
```

The highlight_current_player method resets the background color of all players' buttons and highlights the background of the current player's button .

Python Code

```
def reset_game ( self ):
    self . canvas . delete (" piece ")

    for player in self . players :
            player [" piece "][" position "] = self . get_start_position (
                player [" color "])
            self . draw_piece ( player [" piece "][" position "] , player [" color "])

    self . player_info_label . config ( text =" Game Reset ")

    self . current_player_index = 0
    self . highlight_current_player ()
```

The reset_game method clears all pieces from the canvas, resets player positions, updates the player information label, resets the current player index, and highlights the first player's turn .

Python Code

```
def get_start_position ( self, color ):
    if color == " red ":
            return ( 300, 300 )
    elif color == " green ":
            return ( 400, 400 )
    elif color == " blue ":
            return ( 400, 300 )
    elif color == " yellow ":
            return ( 300, 400 )
```

The get_start_position method returns the starting position for a given player color .

Python Code

```
def check_win_condition ( self ):
    center_position = ( 300, 300 )
    for player in self . players :
```

```
            if player [ 'piece' ][ 'position' ] ==  center_position :
                return True
        return False
```

The check_win_condition method checks if any player has reached the center position, indicating a win condition .

Python Code

```
root = tk . Tk ()
ludo_game = LudoGame ( root )
root . mainloop ()
```

Finally, the main program creates a Tkinter root window, instantiates the LudoGame class with the root window, and starts the Tkinter event loop with root . mainloop (). This loop keeps the GUI application running until the user closes the window .

How To Play Ludo Game

To play the Ludo game created with the provided code, you can follow these instructions :

1. **Run the Code :**
    - Copy the entire provided code into a Python file (e . g . , ludo_game . py).

- Run the script using a Python interpreter .

2. **Game Interface :**

- After running the script, a graphical window will appear with the Ludo board and game controls .

3. **Roll the Dice :**

- Click the " Roll Dice " button to simulate rolling a dice . The button will display the rolled number .

4. **Player Turn :**

- The player with the current turn is highlighted, and their color is displayed on the " Roll Dice " button .

5. **Move Your Piece :**

- The player's piece will move on the board based on the rolled number .
- The game automatically updates the player information label with the current player's turn and the dice roll result .

6. **Winning the Game :**

- The goal is to move your piece to the center of the board (300, 300).
- If a player's piece reaches the center, the game declares that player as the winner .

7. **Reset Game :**

- You can click the " Reset Game " button to start a new game . This will reset the pieces and the player order .

8. **Close the Game :**

- Click the " Close " button to exit the game and close the window .

9. **Repeat Turns :**

- The game follows a turn - based system, and players take turns rolling the dice and moving their pieces .

10. **Enjoy the Game :**

- Have fun playing Ludo with your friends or against computer - controlled players .

Remember that this code provides a basic framework for a Ludo game, and you can customize and extend it according to your preferences . You can add more features, such as player names, sound effects, or additional game logic, to enhance the gaming experience .